Progress®



Progress Data Objects Reference

OpenEdge®

Copyright

© 2021 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Defrag This, Deliver More Than Expected, DevReach (and design), Icenium, Inspec, Ipswitch, iMacros, Kendo UI, Kinvey, MessageWay, MOVEit, NativeChat, NativeScript, OpenEdge, Powered by Chef, Powered by Progress, Progress, Progress Software Developers Network, SequeLink, Sitefinity (and Design), Sitefinity, Sitefinity (and design), SpeedScript, Stylus Studio, Stylized Design (Arrow/3D Box logo), Styleized Design (C Chef logo), Stylized Design of Samurai, TeamPulse, Telerik, Telerik (and design), Test Studio, WebSpeed, WhatsConfigured, WhatsConnected, WhatsUp, and WS_FTP are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries.

Analytics360, AppServer, BusinessEdge, Chef Automate, Chef Compliance, Chef Desktop, Chef Habitat, Chef WorkStation, Corticon.js, Corticon Rules, Data Access, DataDirect Autonomous REST Connector, DataDirect Spy, DevCraft, Fiddler, Fiddler Everywhere, FiddlerCap, FiddlerCore, FiddlerScript, Hybrid Data Pipeline, iMail, JustAssembly, JustDecompile, JustMock, KendoReact, NativeScript Sidekick, OpenAccess, PASOE, Pro2, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, Push Jobs, SafeSpaceVR, Sitefinity Cloud, Sitefinity CMS, Sitefinity Digital Experience Cloud, Sitefinity Feather, Sitefinity Insight, Sitefinity Thunder, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, Supermarket, SupportLink, Unite UX, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

July 2021

Product version: Progress OpenEdge 12.4

Updated: 2021/06/30

Table of Contents

Preface....9

JSDO class and object reference....11

progress.data.JSDO class....12
progress.data.JSDOSession class....25
progress.data.JSRecord class....35
progress.data.PluginManager class....36
request object....37

JSDO properties, methods, and events reference....41

acceptChanges( ) method....44
acceptRowChanges( ) method....47
add( ) method....49
addCatalog( ) method (JSDOSession class)....50
addLocalRecords( ) method....55
addPlugin( ) method....57
addRecords( ) method....61
afterCreate event....64
afterDelete event....67
afterFill event....69
afterInvoke event....70
afterSaveChanges event....72
afterUpdate event....76
assign( ) method (JSDO class)....79
async property....80
authenticationModel property (JSDOSession class)....81
autoApplyChanges property....82
autoSort property....82
batch property....85
beforeCreate event....85
beforeDelete event....87
beforeFill event....88
beforeInvoke event....89
beforeSaveChanges event....90
beforeUpdate event....91
caseSensitive property....93
catalogURIs property....94
clientContextId property....94
connected property....95

data property....96
deleteLocal( ) method....96
fill( ) method....97
find( ) method....104
findById( ) method....106
fnName property....107
foreach( ) method....108
getData( ) method....110
getErrors( ) method....110
getErrorString( ) method....115
getId( ) method....116
getPlugin( ) method....117
getProperties( ) method....118
getProperty( ) method....119
getSchema( ) method....120
getSession( ) stand-alone function....121
hasChanges( ) method....126
hasData( ) method....127
invalidate( ) method....128
invalidateAllSessions( ) stand-alone function....130
invocation method....134
invoke( ) method....136
isAuthorized( ) method....140
jsdo property....143
JSDOs property....143
jsrecord property....143
jsrecords property....144
login( ) method (JSDOSession class)....145
loginHttpStatus property....149
loginResult property....150
logout( ) method (JSDOSession class)....150
name property (JSDO class)....153
name property (JSDOSession class)....154
objParam property....154
offline event....155
online event....157
onOpenRequest property....158
ping( ) method (JSDOSession class)....160
pingInterval property....163
readLocal( ) method....164
record property....165
rejectChanges( ) method....165
rejectRowChanges( ) method....168
remove( ) method....171
response property....172

saveChanges( ) method....174
saveLocal( ) method....188
services property....190
serviceURI property....191
setProperties( ) method....192
setProperty( ) method....193
setSortFields( ) method....194
setSortFn( ) method....195
sort( ) method....197
subscribe( ) method (JSDO class)....200
subscribe( ) method (JSDOSession class)....202
success property....202
table reference property (JSDO class)....203
unsubscribe( ) method (JSDO class)....206
unsubscribe( ) method (JSDOSession class)....207
unsubscribeAll( ) method....208
useRelationships property....208
userName property....209
xhr property....209
Data type mappings for Data Object Services....210
JavaScript data type overview....210
OpenEdge ABL to JavaScript data type mappings....212

Data type mappings for Data Object Services....215

JavaScript data type overview....215
OpenEdge ABL to JavaScript data type mappings....217

Preface

Purpose
This manual describes Progress Data Objects classes, objects, properties, and methods.

Audience
This manual is designed as a reference for developers working with Progress Data Objects.

Documentation conventions
See Documentation Conventions for an explanation of the terminology, format, and typographical conventions used throughout the OpenEdge content library.

1

JSDO class and object reference

The classes and objects described in this reference to support Progress Data Objects comprise the entire implementation for the client-side JavaScript Data Object (JSDO). Each class or object description lists its members by category: properties, methods, and events, with a short description of each member. For a detailed description of each member, see JSDO properties, methods, and events reference on page 41.

Note: JavaScript is a case-sensitive language. So, class type, property, method, and event names, as well as other defined language elements (such as data types) must have the specified letter case.

Note: In addition to the documented members, these classes and objects might also contain undocumented members that are reserved for internal use by OpenEdge.

For details, see the following topics:

- progress.data.JSDO class
- progress.data.JSDOSession class
- progress.data.JSRecord class
- progress.data.PluginManager class
- request object

progress.data.JSDO class

The `progress.data.JSDO` is a JavaScript class that provides access to resources (Data Objects) of a Progress Data Object Service. A single `progress.data.JSDO` object (JSDO instance) provides access to a single resource supported by a given Data Object Service. This resource implements a given Data Object depending on the type of Data Object Service that provides it. For an OpenEdge Data Object Service, the resource is implemented using an ABL Business Entity, which is a singleton class or persistent procedure running on an OpenEdge application server.

Once created for a given Data Object resource, a JSDO provides application-oriented, JavaScript methods that invoke Data Object operations supported by its resource implementation. These can include standard operations for reading and writing data, and can also include custom operations for invoking business logic and more complex data operations, depending on the implementation. An OpenEdge resource maps specified ABL routines to any of several supported operation types, each of which corresponds to a particular method in the JSDO.

For an OpenEdge Data Object Service, you (or another OpenEdge user) identify how a JSDO maps JavaScript methods to operations of a given resource by adding annotations to the singleton procedure or class source code. These annotations map the ABL routines that implement a given resource operation to the corresponding JavaScript method in the JSDO that invokes the operation. You specify these annotations using features of Progress Developer Studio for OpenEdge (Developer Studio) in two ways:

1. When initially creating an ABL Business Entity to implement a Data Object that supports the business operations you want on a specified data model
2. When modifying any existing ABL singleton class or procedure that you want to function as a Business Entity that implements a Data Object resource

You can also use Developer Studio to add a given resource to an existing OpenEdge Data Object Service, as part of generating the Service along with the client-side artifacts required to create a corresponding JSDO for the resource on the client. These client-side artifacts include the Data Service Catalog that defines the schema and operations for each resource you can access in the Service using a corresponding JSDO.

At run time, the JSDO maintains an internal data store (JSDO memory) for managing table records that are exchanged between a given Data Object resource and the JSDO, and it provides methods to read and write data in JSDO memory as individual JavaScript record objects (see the progress.data.JSRecord class on page 35 description). To support this data exchange between a JSDO and its Data Object resource, a given resource can support standard operation types that include Create, Read, Update, Delete (CRUD) operations, and Submit operations. The Read operation of a resource returns a specified result set from the resource to the client; the CUD operations create, update, or delete (respectively) one record at a time in the resource; and a Submit operation creates, updates, and deletes multiple records at a time in the resource. In addition to these standard operations, a resource can also support custom Invoke operations to access ABL business logic in a given OpenEdge resource.

The standard CRUD operations can operate on any OpenEdge single-table resource, or on an OpenEdge multi-table resource (ProDataSet) containing one or more temp-tables. Standard Submit operations are currently available only on an OpenEdge ProDataSet resource that supports before-imaging (with potentially complex server transaction support).

Each standard Data Object operation type (CRUD or Submit) maps to a corresponding standard method of the JSDO on the client. The records of each resource table are managed in the JSDO as an array of record objects, which the standard JSDO methods use to exchange data with the server as supported by the type of Data Object Service. The standard JSDO methods mapped to standard operation types support client access to the common business operations that can be generated for an OpenEdge Business Entity. Other standard methods of the JSDO provide internal access to individual record objects in JSDO memory. Depending on the results of its standard methods, the JSDO also maintains a working record for each table in its memory that you can access directly using table and field references (see the notes in this class description). Thus, using the standard methods of a JSDO and its table references, you can interact with a corresponding Data Object resource in a consistent manner from one JSDO-resource mapping to another.

The JSDO methods to invoke the standard resource operations include `fill( )` to Read resource data and `saveChanges( )` to either Create, Update, or Delete individual records in the resource, or Submit to create, update, and delete multiple records in the resource. Both of these JSDO methods execute asynchronously. Any results, including errors, that are returned from both the resource operations and the JSDO method invocations themselves can be managed using two mechanisms:

1. **Named events** — To which you can subscribe event handlers (callback functions) to process the results when a given event is fired. The available events fire both before and after each resource operation executes, and both before and after execution of the JSDO method that invokes the operation, or operations. The callback for each event has a particular signature that is appropriate for the results returned for the event. See the descriptions of both the JSDO `fill( )` and `saveChanges( )` methods and the associated event descriptions for more information.
2. **jQuery Promises** — Deferred Promise objects, an instance of which can be returned as the method value when results are available. A Promise object provides methods that you can use to register callback functions that, in turn, execute for different conditions associated with the JSDO method results. In order to use jQuery Promises, your development environment must support them (or the exact equivalent). For more information on this mechanism, and how it works with the `fill( )` and `saveChanges( )` methods, see the notes for this class description.

If your environment supports jQuery Promises, you can use either or both of these mechanisms to return results for the standard resource operations.

Like the standard CRUD and Submit operations, a resource can define custom Invoke operations, depending on the type of resource. For OpenEdge resources, an Invoke operation can call and return results for any ABL routine on the server that is defined by the resource. To call an Invoke operation for a specific ABL routine, an OpenEdge resource defines a corresponding *invocation method* that you can call on the JSDO to send the request the server.

Note that data exchanged between server and client using Invoke operations is not automatically stored in JSDO memory. It is initially accessible only from the return values and other results of the invocation methods provided by the JSDO. You can subsequently use other standard JSDO methods to exchange data between the invocation methods and JSDO memory. For example, you can use the JSDO `addRecords( )` method to merge appropriate data returned by an invocation method directly into JSDO memory. The result in JSDO memory can then continue to be maintained and synchronized with the server using the JSDO `fill( )` and `saveChanges( )` methods to invoke the standard CRUD and Submit operations.

Unlike the standard `fill( )` and `saveChanges( )` methods, you can execute custom JSDO invocation methods either synchronously or asynchronously, depending on parameter settings that you use to call them. If you execute an invocation method asynchronously, the JSDO also supports events that fire before and after the method (and its Invoke operation) execute in the resource, returning the results in any event handler callbacks that you subscribe to these events. In addition, you can use the standard JSDO `invoke( )` method as an API to asynchronously call custom invocation methods that return their results in jQuery Promise callbacks, similar to the `fill( )` and `saveChanges( )` methods, as long as your environment supports jQuery Promises.

Note that all JSDO methods that invoke either standard or custom resource operations return a request object to any subscribed event callbacks. This request object supports a number of properties containing the results of the resource operation, some of which appear in the request object only for specific events. Similarly, any Promise object returned by a JSDO method also returns an appropriate request object to each callback that you register for that object. For custom invocation methods that you execute synchronously, this request object is returned as the value of the invocation method. For more information on this request object, see the request object on page 37 description.

When you instantiate a JSDO, it relies on a prior JSDO login session that you can establish using an instance of either the `progress.data.JSDOSession` class or the `progress.data.Session` class. This login session enables optionally secure communications between the client JSDO and the web server, its supported Data Object Services, and ultimately the application server that implements the resource accessed by the JSDO.

A `JSDOSession` insatnce manages its JSDO login session asynchronously using only jQuery Promises to return all session-related results, where as a `Session` instance can manage its JSDO login session either synchronously or asynchronously using several `Session` object events. Note that, similar to managing resource operations using jQuery Promises returned from JSDO methods, you can use the `JSDOSession` class to manage JSDO login sessions only if your environment supports jQuery Promises. Otherwise, you must use the `Session` class to manage the login session for a JSDO. For more information, see the progress.data.JSDOSession class on page 25 description.

Constructors

Two constructors are available for the JSDO. The first constructor takes the name of the corresponding Data Object resource as a parameter; the second constructor takes an initialization object as a parameter.

The resource name specified for the constructor must match the name of a resource provided by a Data Object Service for which a login session has already been established. After the JSDO is created, it uses the information stored in the Data Service Catalog to communicate with the specified resource.

Syntax

```
progress.data.JSDO ( resource-name )
progress.data.JSDO ( init-object )
```

`resource-name`

A `string` expression set to the name of a resource that the JSDO will access. This resource must be provided by a Data Object Service for which a login session has already been established.

`init-object`

An object that can contain any writable JSDO properties. It must contain the required JSDO `name` property, which specifies the Data Object resource that the JSDO will access. This resource must be provided by a Data Object Service for which a login session has already been established.

This *`init-object`* can also contain either or both of the following optional initialization properties:

- **`autoFill`** — A `boolean` that specifies whether the the JSDO invokes its `fill( )` method upon instantiation to initialize JSDO memory with data from its resource. The default value is `false`.
- **`events`** — An object that specifies one or more JSDO event subscriptions, each with its properties represented as an array, with the following syntax:

Syntax

```
events : {
     'event' : [ {
          [ scope : object-ref , ]
          fn : function-ref
     } ] [ ,
     'event' : [ {
          [ scope : object-ref , ]
          fn : function-ref
     } ] ] ...
}
```

event

The name of an event to which the JSDO instance subscribes callbacks. See Events on page 21 for a list of available JSDO events.

object-ref

An optional object reference that defines the execution scope of the callback function invoked when the event fires. If the `scope` property is omitted, the execution scope is the global object (usually the browser or device window).

function-ref

A reference to the callback function that is invoked when the event fires.

Each event passes a fixed set of parameters to its event callback, as described for the event. A reference to the description of each JSDO event is provided later in this class description.

Example

The following example illustrates the use of an initialization object to instantiate a JSDO:

```
dsItems = new progress.data.JSDO({
  name : 'Item',
  autoFill : false,
  events : {
    'afterFill' : [ {
                      scope : this,
                      fn : function (jsdo, success, request) {
                        // afterFill event handler statements ...
                      }
                  } ]
  }
});
```

Properties

Table 1: progress.data.JSDO properties

Member	Brief description (See also the reference entry)
autoApplyChanges property on page 82	A `boolean` on a JSDO that indicates if the JSDO automatically accepts or rejects changes to JSDO memory when you call the `saveChanges( )` method.
autoSort property on page 82	A `boolean` on a JSDO and its table references that indicates if record objects are sorted automatically on the affected table references in JSDO memory at the completion of a supported JSDO operation.
caseSensitive property on page 93	A `boolean` on a JSDO and its table references that indicates if `string` field comparisons performed by supported JSDO operations are case sensitive or case-insensitive for the affected table references in JSDO memory.
name property (JSDO class) on page 153	Returns the name of the Data Object resource for which the current JSDO was created.
record property on page 165	A property on a JSDO table reference that references a `JSRecord` object with the data for the working record of a table referenced in JSDO memory.
table reference property (JSDO class) on page 203	An object reference property on a JSDO that has the name of a corresponding table in the Data Object resource for which the current JSDO was created.
useRelationships property on page 208	A `boolean` that specifies whether JSDO methods that operate on table references in JSDO memory recognize and honor *data-relations* defined in the schema (that is, work only on records of a child table that are related to the working record of a parent table).

Methods

Certain methods of the `progress.data.JSDO` class are called on the JSDO object itself, without regard to a table reference, whether that reference is explicit (specified in the method signature) or implicit (in the case of a JSDO containing only one table that is not explicitly specified). Other methods can be called on a specific table reference that is mapped to one of possibly multiple tables managed by the resource for which the current JSDO is created.

Table 2: progress.data.JSDO class-instance methods

Member	Brief description (See also the reference entry)
acceptChanges( ) method on page 44	Accepts changes to the data in JSDO memory for the specified table reference or for all table references of the specified JSDO.
acceptRowChanges( ) method on page 47	Accepts changes to the data in JSDO memory for a specified record object.
addLocalRecords( ) method on page 55	Reads the record objects stored in the specified local storage area and updates JSDO memory based on these record objects, including any pending changes and before-image data, if they exist.
addRecords( ) method on page 61	Updates JSDO memory with one or more record objects read from an array, single-table, or multi-table resource that are passed in an object parameter, including any pending changes and before-image data, if they exist.
deleteLocal( ) method on page 96	Clears out all data and changes stored in a specified local storage area, and removes the cleared storage area.
fill( ) method on page 97 **Alias:** `read( )`	Initializes JSDO memory with record objects from the data records in a single-table resource, or in one or more tables of a multi-table resource, as returned by the Read operation of the Data Object resource for which the JSDO is created.
getProperties( ) method on page 118	Returns an object containing the names and values of the user-defined properties defined in the current JSDO instance.
getProperty( ) method on page 119	Returns the value of the specified JSDO user-defined property.
hasData( ) method on page 127	Returns `true` if record objects can be found in any of the tables referenced in JSDO memory (with or without pending changes), or in only the single table referenced on the JSDO, depending on how the method is called; and returns `false` if no record objects are found in either case.
hasChanges( ) method on page 126	Returns `true` if JSDO memory contains any pending changes (with or without before-image data), and returns `false` if JSDO memory has no pending changes.
invocation method on page 134	A custom method on the JSDO that executes an Invoke operation defined for a Data Object resource.

Member	Brief description (See also the reference entry)
invoke( ) method on page 136	Asynchronously calls a custom invocation method on the JSDO to execute an Invoke operation defined by a Data Object resource.
readLocal( ) method on page 164	Clears out the data in JSDO memory and replaces it with all the data stored in a specified local storage area, including any pending changes and before-image data, if they exist.
rejectChanges( ) method on page 165	Rejects changes to the data in JSDO memory for the specified table reference or for all table references of the specified JSDO.
rejectRowChanges( ) method on page 168	Rejects changes to the data in JSDO memory for a specified record object.
saveChanges( ) method on page 174	Synchronizes to the server all record changes (creates, updates, and deletes) pending in JSDO memory for the current Data Object resource since the last call to the `fill( )` or `saveChanges( )` method, or since any prior changes have been otherwise accepted or rejected.
saveLocal( ) method on page 188	Saves JSDO memory to a specified local storage area, including pending changes and any before-image data, according to a specified data mode.
setProperties( ) method on page 192	Replaces all user-defined properties in the current JSDO instance with the user-defined properties defined in the specified object.
setProperty( ) method on page 193	Sets the value of the specified JSDO user-defined property.
subscribe( ) method (JSDO class) on page 200	Subscribes a given event callback function to a named event of the current JSDO or table reference.
unsubscribe( ) method (JSDO class) on page 206	Unsubscribes a given event callback function from a named event of the current JSDO or table reference.
unsubscribeAll( ) method on page 208	Unsubscribes all event callback functions from a single named event of the current JSDO, `JSDOSession` or `Session` object, or unsubscribes all event callback functions from all events of the current JSDO, `JSDOSession`, or `Session` object.

Table 3: progress.data.JSDO table-reference methods

Member	Brief description (See also the reference entry)
acceptChanges( ) method on page 44	Accepts changes to the data in JSDO memory for the specified table reference or for all table references of the specified JSDO.
acceptRowChanges( ) method on page 47	Accepts changes to the data in JSDO memory for a specified record object.
add( ) method on page 49 **Alias:** `create( )`	Creates a new record object for a table referenced in JSDO memory and returns a reference to the new record.
addRecords( ) method on page 61	Updates JSDO memory with one or more record objects read from an array, single-table, or multi-table resource that are passed in an object parameter, including any pending changes and before-image data, if they exist.
assign( ) method (JSDO class) on page 79 **Alias:** `update( )`	Updates field values for the specified `JSRecord` object in JSDO memory.
find( ) method on page 104	Searches for a record in a table referenced in JSDO memory and returns a reference to that record if found. If no record is found, it returns `null`.
findById( ) method on page 106	Locates and returns the record in JSDO memory with the internal ID you specify.
foreach( ) method on page 108	Loops through the records of a table referenced in JSDO memory and invokes a user-defined callback function as a parameter on each iteration.
getData( ) method on page 110	Returns an array of record objects for a table referenced in JSDO memory.
getErrors( ) method on page 110	Returns an array of errors from the most recent invocation of Create, Read, Update, Delete, or Submit operations (CRUD or Submit) that you have invoked by calling the JSDO `fill( )` or `saveChanges( )` method on a Data Object resource.
getErrorString( ) method on page 115	Returns any before-image error string in the data of a record object referenced in JSDO memory that was set as the result of a resource Create, Update, Delete, or Submit operation.

Member	Brief description (See also the reference entry)
getId( ) method on page 116	Returns the unique internal ID for the record object referenced in JSDO memory.
getSchema( ) method on page 120	Returns an array of objects, one for each field that defines the schema of a table referenced in JSDO memory.
hasData( ) method on page 127	Returns `true` if record objects can be found in any of the tables referenced in JSDO memory (with or without pending changes), or in only the single table referenced on the JSDO, depending on how the method is called; and returns `false` if no record objects are found in either case.
rejectChanges( ) method on page 165	Rejects changes to the data in JSDO memory for the specified table reference or for all table references of the specified JSDO.
rejectRowChanges( ) method on page 168	Rejects changes to the data in JSDO memory for a specified record object.
remove( ) method on page 171	Deletes the specified table record referenced in JSDO memory.
setSortFields( ) method on page 194	Specifies or clears the record fields on which to automatically sort the record objects for a table reference after you have set its `autoSort` property to `true` (the default).
setSortFn( ) method on page 195	Specifies or clears a user-defined sort function with which to automatically sort the record objects for a table reference after you have set its `autoSort` property to `true` (the default).
sort( ) method on page 197	Sorts the existing record objects for a table reference in JSDO memory using either specified sort fields or a specified user-defined sort function.
subscribe( ) method (JSDO class) on page 200	Subscribes a given event callback function to a named event of the current JSDO or table reference.
unsubscribe( ) method (JSDO class) on page 206	Unsubscribes a given event callback function from a named event of the current JSDO or table reference.

Events

Table 4: progress.data.JSDO events

Member	Brief description (See also the reference entry)
afterCreate event on page 64	Fires after the JSDO, by means of a `saveChanges( )` call following an `add( )` call, sends a request to create a record in the Data Object resource and receives a response to this request from the server.
afterDelete event on page 67	Fires after the JSDO, by means of a `saveChanges( )` call following a `remove( )` call, sends a request to delete a record in the Data Object resource and receives a response to this request from the server.
afterFill event on page 69 **Alias:** `afterRead`	Fires on the JSDO after a call to the `fill( )` method executes and returns the results from the server to JSDO memory for a Read operation request on its Data Object resource.
afterInvoke event on page 70	Fires after a custom invocation method is called asynchronously on a JSDO and a response to the Invoke operation request is received from the server.
afterSaveChanges event on page 72	Fires once for each call to the `saveChanges( )` method on a JSDO, after responses to all create, update, and delete record requests sent to a Data Object resource have been received from the server.
afterUpdate event on page 76	Fires after the JSDO, by means of a `saveChanges( )` call following an `assign( )` call, sends a request to update a record in the Data Object resource and receives a response to this request from the server.
beforeCreate event on page 85	Fires before the JSDO, by means of a `saveChanges( )` call following an `add( )` call, sends a request to create a record in the Data Object resource on the server.
beforeDelete event on page 87	Fires before the JSDO, by means of a `saveChanges( )` call following a `remove( )` call, sends a request to delete a record in the Data Object resource on the server.
beforeFill event on page 88 **Alias:** `beforeRead`	Fires on the JSDO before a call to the `fill( )` method executes and sends a Read operation request to its Data Object resource.
beforeInvoke event on page 89	Fires when a custom invocation method is called asynchronously on a JSDO before the request for the Invoke operation is sent to the server.

Member	Brief description (See also the reference entry)
beforeSaveChanges event on page 90	Fires once on the JSDO before a call to the `saveChanges( )` method sends the first request to create, update, or delete a record in its Data Object resource on the server.
beforeUpdate event on page 91	Fires before the JSDO, by means of a `saveChanges( )` call following an `assign( )` call, sends a request to update a record in the Data Object resource on the server.

The JSDO can subscribe to the events listed in the previous table in either of two ways:

- **Using the JSDO constructor** — In the *`init-object`* parameter of the constructor, list each subscribed event with an optional scope object and an event handler method to be executed when the event fires. See the constructors description for this class.
- **Using the `subscribe( )` method** — See the subscribe( ) method (JSDO class) on page 200.

Note: JSDO events do not fire for custom invocation methods if the method call is synchronous.

Example

The following code fragment shows a simple JavaScript coding sequence for creating and using a JSDO referenced by the variable `dsCustomer` to read resource data, starting with the attempted JSDO login using a `progress.data.JSDOSession` instance:

Table 5: Example — Using a JSDO

```
// create a JSDOSession for a specified web application
// that requires Form authentication
var pdsession = new progress.data.JSDOSession( {
  serviceURI : 'https://testmach:8943/SportsMobile',
  authenticationModel : progress.data.Session.AUTH_TYPE_FORM
});

// log in by authenticating to the specified web application
pdsession.login(username, password).done(
  function( session, result, info ) {
    // login().done handler statements
    // load Catalog for a service that is hosted by the web application
   session.addCatalog('https://testmach:8943/SportsMobile/static/SportsMobileSvc.json'

      ).done(
      function( session, result, details ) {
        // addCatalog().done() handler statements
        // create JSDO for dsCustomer resource
        var dsCustomer = new progress.data.JSDO('dsCustomer');
        // load JSDO memory with resource data
        dsCustomer.fill().done(
          function (jsdo, success, request) {
            // fill().done() handler statements ...
          }).fail(
          function (jsdo, success, request) {
            // fill().fail() handler statements ...
          });
      }).fail(
      function( session, result, details ) {
        // addCatalog().fail() handler statements ...
      });
  }).fail(
  function( session, result, info ) {
    // login().fail() handler statements ...
  });
```

Notes

- Three web-communicating methods of this class, `fill( )`, `saveChanges( )`, and `invoke( )`, always execute asynchronously and return a reference to a jQuery Promise object as the return value when jQuery Promises are supported in the development environment. A Promise is a deferred object that defines methods which, in turn, register callbacks that execute when results are available from the method that returns the Promise. You provide your own code in each callback to handle the results returned for a given Promise method, much like the code you provide in an event callback.

  The primary Promise methods for use with a JSDO object include `done( )`, `fail( )`, and `always( )`, which allow you to separately handle the successful, unsuccessful, and all results (respectively) of a JSDO method execution. Note that for any given JSDO method, the parameter lists for all Promise method callbacks have the same signature. Note also that the callback for the `always( )` method executes with the same values in its parameter list as the callback for the `done( )` or `fail( )` method, whichever one executes with results.

  For more information on the jQuery implementation of Promises and the additional Promise methods that are available to register callbacks, see the following jQuery topics at the specified web locations:

  - **Category: Deferred Object** — http://api.jquery.com/category/deferred-object/
  - **Promise Object** — http://api.jquery.com/Types/#Promise
  - **.promise** — http://api.jquery.com/promise/

- For more information on defining a Data Object resource, including the standard CRUD, Submit, and any custom Invoke operations, for:
  - **OpenEdge Data Object Services** — See the topics on creating Business Entities to implement Data Object resources and updating Business Entities for access by mobile and web apps built using Kendo UI in *OpenEdge Development: Web Services*.

- The JSDO supports a *working record* for each table referenced in JSDO memory. Certain JSDO methods set a specific record as the working record. After other methods execute, there is either no working record set or existing working records remain unchanged. When there is a working record, you can access the fields of the record using one of the following mechanisms:

  Syntax

```
jsdo-ref.table-ref.field-ref
jsdo-ref.field-ref
jsdo-ref.record.data.field-ref // Read-only;
                               // For a jsdo-ref with only one table-ref
jsdo-ref.table-ref.record.data.field-ref // Read-only
jsrecord-ref.data.field-ref // Read-only
```

  jsdo-ref

  The reference to a JSDO, and if the JSDO contains only a single table, an implied reference to any working record that is set for that table.

  table-ref

  A table reference property with the name of a table in *jsdo-ref* memory and a reference to the table working record. There is one table reference in a JSDO for each table referenced by the JSDO. For more information on table reference properties, see the table reference property (JSDO class) on page 203 description.

  field-ref

  A field reference property on a *table-ref* or on the `data` property object, with the name and value of a field in the working record of the referenced table. There is one such field reference and `data` object property for each field defined in the table schema. If *field-ref* is an array field, which is supported in OpenEdge resource tables for example, you can also access individual array elements using either standard JavaScript subscripts or JSDO array-element references in the form, *field-ref_integer*, where *integer* is a one (1)-based integer that qualifies the name of the field reference property that corresponds to the array field element. For more information on field reference properties, see the table reference property (JSDO class) on page 203 description.

  `record`

  A property of type `JSRecord` on a table reference, which references the working record of a referenced table specified by:

  - *jsdo-ref.table-ref*
  - *jsdo-ref* if the JSDO references only one table

  If the JSDO references more than one table, the `record` property is `null` at the JSDO level and is available only on a *table-ref*.

`data`

A property on a `JSRecord` object with the field values for the working record specified by:

- *`jsdo-ref.table-ref`*
- *`jsdo-ref`* if the JSDO references only one table
- A *`jsrecord-ref`* returned for an associated JSDO table reference

Note: If a *`field-ref`* has the same name as a built-in property or method of the JSDO, you must use the `data` property to reference its value in the working record.

Caution: Never write directly to a *`field-ref`* using this `data` property; in this case, use *`field-ref`* only to read the data. Writing field values using the `data` property does not mark the record for update when calling the `saveChanges( )` method, nor does it re-sort the record in JSDO memory according to any order you have established using the `autoSort` property. To mark a record for update and automatically re-sort the record according to the `autoSort` property, you must assign a field value either by setting a *`jsdo-ref.table-ref.field-ref`* for a working record or by calling the `assign( )` method on a valid *`table-ref`* or `JSRecord` object reference.

`jsrecord-ref`

A reference to a `JSRecord` object for a table referenced in JSDO memory. You can return a *`jsrecord-ref`* for a working record as the value of the `record` property or as a value returned by supported JSDO built-in methods that return a working record, such as `add( )` and `find( )`.

For more information on properties available to reference working record fields using this syntax, see the properties listed in this class description and in the description of the progress.data.JSRecord class on page 35. For more information on the methods for setting the working record for referenced tables, see the methods listed in this class description and in the description of the progress.data.JSRecord class on page 35.

- Many standard JSDO methods are actually invoked on a JSDO table reference, and can only be invoked on the JSDO itself when its JSDO memory is initialized with a single table.
- For a multi-table resource, such as an OpenEdge ProDataSet, the JSDO accesses the data for all unrelated tables in JSDO memory as top-level tables of the JSDO. Access to data for all related child tables depends on the working record of the parent table in the JSDO and the setting of the `useRelationships` property.

See also

progress.data.JSRecord class on page 35, record property on page 165, table reference property (JSDO class) on page 203

progress.data.JSDOSession class

The `progress.data.JSDOSession` is a JavaScript class that provides methods, properties, and events to create and manage a JSDO login session. A *JSDO login session* includes a single end point (web application) and a single authentication model (Anonymous, HTTP Basic, or HTTP Form), and manages user access to Progress Data Object resources using instances of the `progress.data.JSDO` class (JSDO). This includes loading the definitions for the Data Object resources that JSDOs can access and starting and managing JSDO login sessions on web servers that provide access to these resources.

This JSDOSession class supports methods for loading Data Service Object resource definitions and managing login sessions only run asynchronously and return jQuery Promises to handle the results in callbacks registered using Promise methods. The information required to instantiate and start JSDO login sessions with JSDOSession objects is handled somewhat differently than with Session objects, including optional support for web app page refresh.

Note: If your development environment does not support jQuery Promises (or the exact equivalent), you must use the progress.data.Session class to manage login sessions for JSDOs.

Like a Session object, a JSDOSession object manages user authentication and session identification information in HTTP/S messages sent between JSDOs running in a mobile app and Data Object Services running on a web server, each of which provide access to a set of Data Object resources. The authentication information includes a username and password (*user credentials*), if necessary, to authenticate a JSDO login session in a web application, which provides a REST transport between a set of Data Object Services and any client mobile app that accesses them. The session identification information includes the URI for the web application and might include a session ID that identifies the JSDO login session and helps to manage interactions between the client and the web application.

To start a JSDO login session for a JSDOSession object to manage, first instantiate the JSDOSession for a particular web application URI and authentication model as required by the web application's security configuration. Then invoke the object's login( ) method. Pass the login( ) method any required user credentials to authenticate JSDOSession access to the web application according to its authentication model. Once started, a login session for a web application supports all Data Object Services that the application provides, each of which can define one or more resources for access by JSDOs.

Each Data Object Service provides a separate JSON file (Data Service Catalog) that defines the schema for its set of resources and the operations to communicate between these resources and the JSDO instances that access them from a mobile app. To create a JSDO instance for a Data Object resource, a JSDOSession object must first load the Catalog that defines the resource using its addCatalog( ) method. This method can load this Catalog from the web application, from some other web location, or from a location on the client where the mobile app runs, and the method can also accept credentials to access a Data Service Catalog that is protected separately from the web application itself.

Note: The standard mechanism supported by the JSDO framework to invoke most of the features of instantiating a JSDOSession object, establishing its JSDO login session, and loading a Data Service Catalog, is to invoke a single call to the progress.data.getSession( ) stand-alone function. For more information, see the getSession( ) stand-alone function on page 121 description. Also, see the Notes on page 32 on this class.

Once a Data Service Catalog is loaded into a JSDOSession object, you can instantiate a JSDO to access a particular resource defined in that Catalog. Once the JSDOSession is logged into its web application, the JSDO can then access its resource, relying on any authentication information for the login session (if necessary) to authorize this access.

Multiple JSDOs can thus rely on a single JSDOSession object to manage session access to all Data Object Services and their resources defined by the same web application.

Constructor

Instantiates a JSDOSession object that you can use to start and manage a JSDO login session in a web application and load the Data Service Catalog for each supported Data Object Service whose resources are accessed using JSDOs.

Syntax

```
progress.data.JSDOSession ( options )
```

options

An object that contains the following configuration properties:

- `serviceURI` — (Required) A `string` expression containing the URI of the web application for which to start a JSDO login session. This web application must support one or more Data Object Services in order to create JSDOs for the resources provided by the application. This URI is appended with a string that identifies a resource to access as part of the login process.

  If the mobile app from which you are logging in is a hybrid app that will be installed to run directly in a native device container, or if it is a web app deployed to a different web server from the web application that hosts the JSDO login session, you must specify `serviceURI` as an absolute URI to the Tomcat server domain or host and port, for example, `http://www.progress.com/SportsMobileApp`, or perhaps for testing, `http://testmach:8980/SportsMobileApp`.

  If the mobile app from which you are logging in is a web app deployed to the same Apache Tomcat server as the web application that hosts the JSDO login session, you can specify `serviceURI` as a relative URI, for example, `/SportsMobileApp`, which is relative to the deployment end point (Tomcat server domain or host and port).

  Note: Once the `login( )` method executes, the value you pass for `serviceURI` also sets the value of the `sessionURI` property on the current `JSDOSession` object, whether or not JSDO login completes successfully.

- `authenticationModel` — (Optional) A `string` constant that specifies one of the three authentication models that the `JSDOSession` object supports, depending on the web application configuration:

  - `progress.data.Session.AUTH_TYPE_ANON` — (Default) The web application supports Anonymous access. No authentication is required.
  - `progress.data.Session.AUTH_TYPE_BASIC` — The web application supports HTTP Basic authentication and requires a valid username and password. To have the `JSDOSession` object manage access to the web application's resources for you, you need to pass these credentials in a call to the `JSDOSession` object's `login( )` method. Typically, you would require the user to enter their credentials into a login dialog provided by your mobile app, either using a form of your own design or using a template provided by Progress Software Corp.
  - `progress.data.Session.AUTH_TYPE_FORM` — The web application uses Form-based authentication. Like HTTP Basic, Form-based authentication requires user credentials for access to protected resources; the difference is that the web application itself sends a form to the client to get the credentials. However, when you have the `JSDOSession` object manage access to the web application's resources, you handle Form-based authentication the same way that you handle Basic—get the user's credentials yourself and pass them to the `login( )` method. The `JSDOSession` intercepts the form sent by the web application and handles the authentication without that form being displayed.

  If the web application requires authentication, you must set this value correctly to ensure that users can log in.

  For more information on these authentication models and how to configure them for a web application, see the sections on configuring web server authentication models in the administration documentation for the server hosting your Data Object Service.

- name — (Optional) A string with an operative value that you define to enable page refresh support for a JSDOSession object when it is instantiated for access by a client web app. The operative value can be any value other than the empty string (""), null, or undefined.

  If this page refresh support is enabled, when the web app successfully logs the newly instantiated JSDOSession into its server web application, the object stores the state of its JSDO login session using this name property value as a key. Then, at any time after log in, if the user initiates a browser refresh on the web app page, the JSDOSession object automatically identifies and restores its login session using this value. This helps to ensure, after a page refresh, that the web app does not need to prompt the user again for credentials in order to re-establish its connection to the web application for this JSDOSession.

  If you do not specify an operative value for name (the default), or you specify a non-operative value, such as the empty string (""), null, or undefined, the JSDOSession is instantiated without this page refresh support.

  For more information on how a login session is restored for a JSDOSession object with page refresh support, see the Notes on page 32 for this class description.

  Note: If you pass the JSDOSession constructor the same value for name as an existing JSDOSession, it will return a JSDOSession using that same key. Both of them have the potential to overwrite each other. You must ensure that you pass a unique name value for each call to the JSDOSession constructor (or the getSession( ) stand-alone function).

  Note: Page refresh supports **only** Anonymous and Form-based authentication. You cannot enable page refresh support when you set authenticationModel for a JSDOSession to progress.data.Session.AUTH_TYPE_BASIC; in this case, any setting of the name property is ignored. To enable page refresh support, you must set the authenticationModel property to either progress.data.Session.AUTH_TYPE_ANON or progress.data.Session.AUTH_TYPE_FORM.

  Note: To help manage the results of a page refresh for any web app that accesses a JSDO, the JSDOSession class provides an isAuthorized( ) method that you can call to test that an authorized login session is established for a given JSDOSession object. For more information and an example, see the description of the isAuthorized( ) method on page 140.

Note: The standard mechanism supported by the JSDO framework to invoke most of the features of instantiating a JSDOSession object, establishing its JSDO login session, and loading a Data Service Catalog, is to invoke a single call to the progress.data.getSession( ) stand-alone function. For more information, see the getSession( ) stand-alone function on page 121 description. Also, see the Notes on page 32 on this class.

Properties

Table 6: progress.data.JSDOSession properties

Member	Brief description (See also the reference entry)
authenticationModel property (JSDOSession class) on page 81	Returns a string constant that was passed as an option to the object's class constructor, and specifies the authentication model that the current JSDOSession object requires to start a JSDO login session in the web application for which the JSDOSession object was also created.

Member	Brief description (See also the reference entry)
catalogURIs property on page 94	Returns the list of URIs successfully used to load Data Service Catalogs into the current `JSDOSession` object.
clientContextId property on page 94	The value of the most recent client context identifier (CCID) that the current `JSDOSession` object has found in the `X-CLIENT-CONTEXT-ID` HTTP header of a server response message.
connected property on page 95	Returns a `boolean` that indicates the most recent online status of the current `JSDOSession` object when it last determined if the web application it manages was available.
JSDOs property on page 143	Returns an array of JSDOs that use the current `JSDOSession` object to communicate with their Data Object Services.
loginHttpStatus property on page 149	Returns the specific HTTP status code returned in the response from the most recent login attempt on the current `JSDOSession` object.
loginResult property on page 150	Returns the return value of the `login( )` method, which is the basic result code for the most recent login attempt on the current `JSDOSession` object.
name property (JSDOSession class) on page 154	Returns the value of any `name` property to enable page refresh support that was passed to the constructor of the current `JSDOSession` object.
onOpenRequest property on page 158	Returns the reference to a user-defined callback function that the `JSDOSession` object executes to modify a request object before sending the request object to the server.
pingInterval property on page 163	A `number` that specifies the duration, in milliseconds, between one automatic execution of the current `JSDOSession` object's `ping( )` method and the next.
services property on page 190	Returns an array of objects that identifies the Data Object Services that have been loaded for the current `JSDOSession` object and its web application.

Member	Brief description (See also the reference entry)
serviceURI property on page 191	Returns the URI to the web application that has been passed as an option to the class constructor for the current JSDOSession object.
userName property on page 209	Returns the username passed as a parameter to the most recent call to the login( ) method on the current JSDOSession object

Methods

Table 7: progress.data.JSDOSession class-instance methods

Member	Brief description (See also the reference entry)
addCatalog( ) method (JSDOSession class) on page 50	Loads one or more local or remote Data Service Catalogs into the current JSDOSession object.
invalidate( ) method on page 128	Terminates the login session managed by the current JSDOSession object and permanently disables the object, rendering it unable to start a new login session.
isAuthorized( ) method on page 140	Determines if the current JSDOSession object has authorized access to the web application specified by its serviceURI property setting.
login( ) method (JSDOSession class) on page 145	Starts a JSDO login session in a web application for the current JSDOSession object by sending an HTTP request with specified user credentials to the web application URI specified in the object's constructor.
logout( ) method (JSDOSession class) on page 150	Terminates the login session on the web application managed by the current JSDOSession object and leaves the object available to start a new login session with a call to its login( ) method.
ping( ) method (JSDOSession class) on page 160	Determines the online state of the current JSDOSession object from its ability to access the web application that it manages, and for an OpenEdge web application, from detecting if its associated application server is running.
subscribe( ) method (JSDOSession class) on page 202	Subscribes a given event callback function to an event of the current JSDOSession object.

Member	Brief description (See also the reference entry)
unsubscribe( ) method (JSDOSession class) on page 207	Unsubscribes a given event callback function from an event of the current JSDOSession object.
unsubscribeAll( ) method on page 208	Unsubscribes all event callback functions from a single named event of the current JSDO, JSDOSession or Session object, or unsubscribes all event callback functions from all events of the current JSDO, JSDOSession, or Session object.

Events

Table 8: progress.data.JSDOSession events

Member	Brief description (See also the reference entry)
offline event on page 155	Fires when the current JSDOSession object detects that the device on which it is running has gone offline, or that the web application to which it has been connected is no longer available.
online event on page 157	Fires when the current JSDOSession object detects that the device on which it is running has gone online after it was previously offline, or that the web application to which it is connected is now available after it was previously unavailable.

Example — Using the JSDOSession class

This example shows how you might create a JSDOSession object using the URI and authentication model of a web application (CustService), log into the web application,and load the Data Service Catalog for a Data Object Service provided by that web application. It creates this login session for a JSDO that is created and accessed by an instance of the JSDO dialect of the Kendo UI DataSource (dataSource), which provides a Customer resource from the Data Object Service for access by a Kendo UI Grid:

```
var session;

/* create jsdoSession   */
session = new progress.data.JSDOSession(
        { serviceURI: "http://localhost:8980/CustService",
          authenticationModel: progress.data.Session.AUTH_TYPE_FORM
        };

/* create login screen (using UI defined in HTML file) */
window.loginView = kendo.observable( {
    submit: function() {
        var loginParams = {
                username: this.username,
                password: this.password) };

        /* log in (also loads CustService.json if it succeeds) */
        session.login(loginParams
            ).done( // Logged in
            function(session, result, info) {
                session.addCatalog("http://localhost/.../CustService.json"
                    ).done( // Data Service Catalog loaded
```

```
                    function(session, result, details){
                        /* success function – create grid and datasource */
                        $("#grid").kendoGrid( {
                            dataSource: {
                                type: "jsdo",
                                transport: {
                                    jsdo: { resourceName: "Customer" }
                                },
                                error: function(e) {
                                    console.log("Error: ", e);
                                }
                            },
                            /* etc., other kendoGrid properties */
                        };

                        /* switch UI to show grid */
                        window.location.href = "#grid";
                    }).fail( // Data Service Catalog not loaded
                    function(session, result, details){
                        alert("Data Service Catalog failed to load");
                }); //JSDOSession addCatalog()

            }).fail( // Not logged in
            function(session, result, info) {
                /* display error message, stay on login screen */
                alert("Login failed");
        }); // JSDOSession login()
    } // observable submit method
} ); // kendo.observable
```

The JSDO automatically finds and uses the `JSDOSession` object where its `Customer` resource is loaded from the Data Service Catalog.

Notes

- The six web-communicating methods of this class, `invalidate( )`, `isAuthorized( )`, `login( )`, `addCatalog( )`, `logout( )`, and `ping( )`, as well as the stand-alone functions, `progress.data.getSession( )` and `progress.data.invalidateAllSessions( )`, all execute asynchronously and return a reference to a jQuery Promise object as the return value when jQuery Promises are supported in the development environment. A Promise is a deferred object that defines methods which, in turn, register callbacks that execute when results are available from the method that returns the Promise. You provide your own code in each callback to handle the results returned for a given Promise method, much like the code you provide in an event callback.

  The primary Promise methods for use with a `JSDOSession` object include `done( )`, `fail( )`, and `always( )`, which allow you to separately handle the successful, unsuccessful, and all results (respectively) of a `JSDOSession` method execution. Note that for any given `JSDOSession` method, the parameter lists for all Promise method callbacks have the same signature. Note also that the callback for the `always( )` method executes with the same values in its parameter list as the callback for the `done( )` or `fail( )` method, whichever one executes with results.

  For more information on the jQuery implementation of Promises and the additional Promise methods that are available to register callbacks, see the following jQuery topics at the specified web locations:

  - **Category: Deferred Object** — http://api.jquery.com/category/deferred-object/
  - **Promise Object** — http://api.jquery.com/Types/#Promise
  - **.promise** — http://api.jquery.com/promise/

- When you have the credentials from a user login, you have two options to create and initialize a JSDO login session:

1. As shown in the Example — Using the JSDOSession class on page 31, manually instantiate the `JSDOSession` object for a specified network server (web application) and if successful, invoke its `login( )` method to start the associated JSDO login session and if successful, invoke its `addCatalog( )` method to load one or more Data Service Catalogs and if successful, create one or more JSDO instances to access the resources provided by the initialized Data Object Services.
2. Make one call to the `progress.data.getSession( )` stand-alone function (the standard mechanism provided by the JSDO framework to start a login session), which automatically instantiates the `JSDOSession` for access to a specified web application, and if successful, invokes its `login( )` method to start the login session, and if successful, invokes its `addCatalog( )` method to load a single Data Service Catalog. If the `getSession( )` invocation is successful, you can then create one or more JSDO instances to access resources provided by the single specified Data Object Service. If you require access to additional Data Object Services, you can also make additional calls to `addCatalog( )` to load the Catalogs. If you require access to additional web applications on this network server (or on other network servers, with CORS support), you can make additional calls to `getSession( )` to create as many login sessions as necessary.

At any point during an established login session, you can call the `ping( )` method to determine if its `JSDOSession` object is in an online state and able to access both its web application and associated application server.

Caution: Under certain conditions when a server error occurs, either or both the asynchronous `login( )` and `addCatalog( )` methods fail to return a result and notify the app that the error has occurred. However, both of these methods provide an option (which is set by default) that can force these methods to return a result after a specified timeout threshold.

Following is a known set of inclusive conditions that, if all are true, prevent the asynchronous `login( )` and `addCatalog( )` methods from returning an error result to the app:

- The app is running on an iOS device.
- The failure is caused by invalid credentials.
- The authentication model used to authenticate the credentials is HTTP Basic.

The option provided to force these methods to return a result is an `iOSBasicAuthTimeout` property that you can pass to each method in an object parameter. This property specifies a timeout threshold value that when reached forces the method to return an error result to the app in any registered Promise method callback that executes in response to an error result, such as the `fail( )`callback. The session itself enforces a default timeout threshold for these methods of four (4) seconds, which you can change by setting this property to a different value. A value of zero (0) cancels the timeout threshold entirely, which prevents the method from returning under the specified conditions. Note that any non-zero timeout threshold is only operative for these methods when all of the following conditions are true:

- The JSDO session uses Basic authentication.
- The app is running on an iOS device (iPhone, iPad, or iPod), either as a hybrid app or as a web app in a browser (such as Safari).

Otherwise, no timeout is in effect for either of these methods (and should not be necessary), even if you explicitly set this property to a threshold value in its object argument before invoking the method.

- When your mobile or web app user has signaled their intention to logout, and if necessary for your application, you have three options to terminate one or more current JSDO login sessions, with the following trade-offs:

  1. Invoke the `logout( )` method on each `JSDOSession` object. This both terminates the login session and allows the `JSDOSession` to start another login session by calling `login( )` on it again. Note that

this option can pose a security risk and is not recommended, especially for secure logins (other than Anonymous).

2. Invoke the `invalidate( )` method on each `JSDOSession` object. This both terminates the login session and disables the `JSDOSession` from any further access, preventing any attempt to start another login session with it.
3. Invoke the `progress.data.invalidateAllSessions( )` stand-alone function. This invokes `invalidate( )` on every `JSDOSession` object that you created using the `progress.data.getSession( )` stand-alone function and is currently managing a login session for the app. This is a convenient way to terminate multiple login sessions unless you have a reason to retain one or more of them for further use, such as Anonymous login sessions you might be using to look-up read-only information. In that case, you must call `invalidate( )` or `logout( )` only for each session you want to terminate.

Note: If you did not use `getSession( )` to create a `JSDOSession` object, but instantiated the class manually, you can only log out its session or invalidate the object permanently by calling `logout( )` or `invalidate( )` on the `JSDOSession` instance manually.

- The behavior of a login session using this class depends on the authentication model of the web server and how its resources are protected and accessed. For more information, see the description of the `login( )` method.
- If you have special requirements for sending `JSDOSession` requests to the web server, you can modify the XMLHttpRequest object that is sent by the `JSDOSession` object. To do so, assign a callback function as the value of `onOpenRequest` property.
- You can enable page refresh support accessed by a client web app when you create a `JSDOSession` instance using either the class constructor or the `progress.data.getSession( )` stand-alone function. If you enable page refresh support, when the user initiates a browser page refresh on the web app page and you instantiate the corresponding `JSDOSession` object with the same `name` property value, the following `JSDOSession` properties will have the same values as they did for the corresponding `JSDOSession` instance before the page refresh was initiated (with one exception):
  - `loginResult`
  - `clientContextId`
  - `connected`
  - `loginHttpStatus`
  - `userName`
  - `pingInterval`
  - `authenticationModel` (passed to the constructor anyway)
  - `serviceURI` (passed to the constructor anyway)

  The one exception is if the values of the `authenticationModel` and `serviceURI` properties passed to the `JSDOSession` constructor are different from existing saved values. In this case, the saved data is ignored and cleared out, and a subsequent invocation of the `isAuthorized( )` method returns `progress.data.Session.LOGIN_AUTHENTICATION_REQUIRED` as the *`result`* parameter value to any callback registered for the Promise that is returned by the method.

See also

addCatalog( ) method (JSDOSession class) on page 50, getSession( ) stand-alone function on page 121, invalidate( ) method on page 128, invalidateAllSessions( ) stand-alone function on page 130, isAuthorized( ) method on page 140, login( ) method (JSDOSession class) on page 145, logout( ) method (JSDOSession class) on page 150, onOpenRequest property on page 158, ping( ) method (JSDOSession class) on page 160, progress.data.JSDO class on page 12

progress.data.JSRecord class

The `progress.data.JSRecord` is a JavaScript class that represents a record instance for any table stored in the JSDO memory of an associated `progress.data.JSDO` class instance (JSDO). The constructor for `progress.data.JSRecord` is protected. `JSRecord` instances are created internally by JSDO methods.

Properties

Table 9: progress.data.JSRecord class properties

Member	Brief description (See also the reference entry)
data property on page 96	The data (field) and state values for a record associated with a `JSRecord` object.

Methods

Table 10: progress.data.JSRecord class methods

Member	Brief description (See also the reference entry)
acceptRowChanges( ) method on page 47	Accepts changes to the data in JSDO memory for a specified record object.
assign( ) method (JSDO class) on page 79 **Alias:** `update( )`	Updates field values for the specified `JSRecord` object in JSDO memory.
getErrorString( ) method on page 115	Returns any before-image error string in the data of a record object referenced in JSDO memory that was set as the result of a resource Create, Update, Delete, or Submit operation.
getId( ) method on page 116	Returns the unique internal ID for the record object referenced in JSDO memory.
remove( ) method on page 171	Deletes the specified table record referenced in JSDO memory.
rejectRowChanges( ) method on page 168	Rejects changes to the data in JSDO memory for a specified record object.

Events

This class has no events.

Example

The following example assumes that a JSDO is referenced by the `jsdo` variable, and that an instance associated with that JSDO is referenced by the `uihelper` variable. The example creates a new record object and displays it, along with a message with credit information using properties of the record object:

```
function addRecord() {
  var jsrecord = jsdo.add( {Balance: 10000, State: 'MA'} );
  uihelper.display( );
  alert( 'Record ID: ' + jsrecord.getId( ) + ' CreditLimit: ' +
         jsrecord.data.CreditLimit );
}
```

Notes

Using the `add( )`, `find( )`, `findById( )`, or `foreach( )` method, or the `record` property, on a given JSDO and table reference, a `JSRecord` instance returns a working record for the table referenced in JSDO memory. You can then use properties and methods of the `JSRecord` to update, delete, or display the specified record from the JSDO.

See also

add( ) method on page 49, find( ) method on page 104, findById( ) method on page 106, foreach( ) method on page 108, progress.data.JSDO class on page 12, record property on page 165, table reference property (JSDO class) on page 203

progress.data.PluginManager class

The `progress.data.PluginManager` is a JavaScript class that provides class-level methods to create and manage custom JSDO plugins. A *JSDO plugin* is an object with functions that execute for any supported Data Object operation that you invoke on a client JSDO instance. You can define these functions for a given plugin to customize the default behavior of the associated operation.

To use a JSDO plugin for a Data Object operation you must complete the following tasks:

- For an OpenEdge Data Object, annotate the ABL Business Entity that implements the operation to identify the name of the JSDO plugin with the operation in the Data Service Catalog.
- Use the (class-level) methods of this class to add (*register*) a new JSDO plugin or modify an existing plugin for use by any client JSDO instance that invokes the operation on the server.

For more information on completing these tasks, see the descriptions of the methods of this class.

Currently, the only Data Object operation that supports JSDO plugins is the Read operation (`fill( )` method), and the only supported plugin type is a *MappingType* plugin that allows you to customize both the operation request and the response returned from executing the operation on the server.

The JSDO also defines a single built-in MappingType plugin with the reserved name, `"JFP"`. This built-in plugin supports client handling of requests for Read operations that recognize JSON Filter Pattern (JFP) input. For more information on JFP input and this built-in MappingType plugin, see the description of the fill( ) method on page 97. You can customize the default request handling in this built-in `"JFP"` plugin, as well as add a custom response handler to it. You can also register your own plugins that are based on this built-in plugin (or that are based on any custom plugin previously registered in the document context).

This class has no constructor, but provides class-level methods to manage plugin registration.

Methods

Table 11: progress.data.PluginManager class-level methods

Member	Brief description (See also the reference entry)
addPlugin( ) method on page 57	Adds (registers) a JSDO plugin object with a specified name in the current document context that can be used with a supported Data Object operation invoked on a JSDO instance.
getPlugin( ) method on page 117	Returns a reference to the registered JSDO plugin object with the specified name.

Note

For more information on registering a new custom JSDO plugin or on customizing a previously registered JSDO plugin, as well as on specifying a JSDO plugin for use by a Data Object operation on the server, see the description of the addPlugin( ) method on page 57.

See also

fill( ) method on page 97

request object

An object containing data and status information returned from a call to one of the methods of an associated `progress.data.JSDO` (JSDO) instance that executes a CRUD, Submit, or Invoke operation on a resource of a Data Object Service. This `request` object is returned by the associated JSDO method call: `fill( )`, `saveChanges( )`, or a given invocation method.

In the case of an asynchronous call (any resource operation that you execute asynchronously), the `request` object is passed as a parameter to any event callback functions that you subscribe to associated JSDO events, or to any callback functions that you register using a returned jQuery Promise object. For Invoke operations that you execute synchronously, the object is available as the return value of the corresponding JSDO invocation method. The object is also passed as a parameter to any event callback functions that you subscribe to the `online` and `offline` events of the `JSDOSession` or `Session` object that manages Data Object Services for the JSDO.

Properties

Table 12: Request object properties

Member	Brief description (See also the reference entry)
async property on page 80	A `boolean` that indicates, if set to `true`, that the Data Object resource operation was executed asynchronously in the mobile app.
batch property on page 85	A reference to an object with a property named `operations`, which is an array containing the request objects for each of the one or more record-change operations on a resource performed in response to calling the JSDO `saveChanges( )` method without using Submit (either with an empty parameter list or with the single parameter value of `false`).
fnName property on page 107	For an Invoke operation, the name of the custom JSDO invocation method that executed the operation.
jsdo property on page 143	An object reference to the JSDO that performed the operation returning the request object.
jsrecord property on page 143	A reference to the record object that was created, updated, or deleted by the current record-change operation.
jsrecords property on page 144	An array of references to record objects that are created, updated, or deleted on the server for a Submit operation (invoking `saveChanges(true)`) on a Data Object resource that supports before-imaging.
objParam property on page 154	A reference to the object, if any, that was passed as an input parameter to the JSDO method that has returned the current request object.
response property on page 172	Returns an object or `string` containing data and status information from an operation invoked on a Data Object resource.
success property on page 202	A `boolean` that when set to `true` indicates that the Data Object resource operation was successfully executed.
xhr property on page 209	A reference to the XMLHttpRequest object used to make an operation request on a resource of a Data Object Service.

Methods

This object has no methods.

Events

This object has no events.

See also

fill( ) method on page 97, invocation method on page 134, progress.data.JSDO class on page 12, progress.data.JSDOSession class on page 25, saveChanges( ) method on page 174

2

JSDO properties, methods, and events reference

This reference describes the properties, methods, and events of the JavaScript classes and objects described in JSDO class and object reference on page 11. Where a given member can be accessed on different object types, the member description indicates this in an **Applies to** list.

If multiple object types support access to the same member, the member description indicates any differences in behavior. For example, for methods, the method syntax shows how to call the method on each object type, and the description indicates when to call the method, and to what effect, for each object type where you can call it.

For details, see the following topics:

- acceptChanges( ) method
- acceptRowChanges( ) method
- add( ) method
- addCatalog( ) method (JSDOSession class)
- addLocalRecords( ) method
- addPlugin( ) method
- addRecords( ) method
- afterCreate event
- afterDelete event
- afterFill event

- afterInvoke event
- afterSaveChanges event
- afterUpdate event
- assign( ) method (JSDO class)
- async property
- authenticationModel property (JSDOSession class)
- autoApplyChanges property
- autoSort property
- batch property
- beforeCreate event
- beforeDelete event
- beforeFill event
- beforeInvoke event
- beforeSaveChanges event
- beforeUpdate event
- caseSensitive property
- catalogURIs property
- clientContextId property
- connected property
- data property
- deleteLocal( ) method
- fill( ) method
- find( ) method
- findById( ) method
- fnName property
- foreach( ) method
- getData( ) method
- getErrors( ) method
- getErrorString( ) method
- getId( ) method
- getPlugin( ) method
- getProperties( ) method
- getProperty( ) method

- getSchema( ) method
- getSession( ) stand-alone function
- hasChanges( ) method
- hasData( ) method
- invalidate( ) method
- invalidateAllSessions( ) stand-alone function
- invocation method
- invoke( ) method
- isAuthorized( ) method
- jsdo property
- JSDOs property
- jsrecord property
- jsrecords property
- login( ) method (JSDOSession class)
- loginHttpStatus property
- loginResult property
- logout( ) method (JSDOSession class)
- name property (JSDO class)
- name property (JSDOSession class)
- objParam property
- offline event
- online event
- onOpenRequest property
- ping( ) method (JSDOSession class)
- pingInterval property
- readLocal( ) method
- record property
- rejectChanges( ) method
- rejectRowChanges( ) method
- remove( ) method
- response property
- saveChanges( ) method
- saveLocal( ) method

- services property
- serviceURI property
- setProperties( ) method
- setProperty( ) method
- setSortFields( ) method
- setSortFn( ) method
- sort( ) method
- subscribe( ) method (JSDO class)
- subscribe( ) method (JSDOSession class)
- success property
- table reference property (JSDO class)
- unsubscribe( ) method (JSDO class)
- unsubscribe( ) method (JSDOSession class)
- unsubscribeAll( ) method
- useRelationships property
- userName property
- xhr property
- Data type mappings for Data Object Services

acceptChanges( ) method

Accepts changes to the data in JSDO memory for the specified table reference or for all table references of the specified JSDO.

If the method succeeds, it returns `true`. Otherwise, it returns `false`.

Note: This method is most useful when the JSDO `autoApplyChanges` property is set to `false`. In this case, you typically invoke this method **after** calling the `saveChanges( )` method in order to accept a series of changes after they have been successfully applied to the server. If the `autoApplyChanges` property is `true`, the JSDO automatically accepts or rejects changes for the specified table reference, or for all table references of the specified JSDO, based on the success of the corresponding record-change operations on the server resource.

Note: Accepting all pending changes in JSDO memory—or even pending changes for a single table reference—because none raised an error from the server might be too broad an action for your application. If so, consider using `acceptRowChanges( )` to accept changes a single table record at a time. For more information, see the description of `acceptRowChanges( )` method.

Return type: `boolean`

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Working record: The working record is set depending on the changes accepted.

Syntax

```
jsdo-ref.acceptChanges ( )
jsdo-ref.table-ref.acceptChanges ( )
```

jsdo-ref

A reference to the JSDO. If you call the method on *jsdo-ref*, the method accepts changes for all table references in the JSDO.

table-ref

A table reference on the JSDO. If you call the method on *table-ref*, the method accepts changes for the specified table reference.

When you accept changes on a table reference, this method makes the record objects in the specified table reflect all pending changes to them in JSDO memory. When you accept changes on the JSDO reference, the method makes the record objects in all tables of the JSDO reflect all pending changes to them in JSDO memory. As the specified changes are accepted, the method also empties any associated before-image data, clears all associated error message settings, and removes the associated record change indications from JSDO memory.

Note: Regardless if you call `acceptChanges( )`, any error message settings that result from Data Object resource operations invoked by the most recent execution of the `fill( )` or `saveChanges( )` method remain available for return by the `getErrors( )` method until the next execution of either `fill( )` or `saveChanges( )`. For more information, see the getErrors( ) method on page 110 description.

This method is especially useful for a resource that supports before-imaging (such as an OpenEdge ProDataSet) and handles the results of all record changes that are sent using a Submit operation as part of a single server transaction that successfully commits all the changes. When the Submit operation returns successfully, calling this method ensures that JSDO memory is synchronized with all the record changes that were committed in the server transaction.

Note: After this method accepts changes, and if you have set up automatic sorting using the `autoSort` property, all the record objects for affected table references are sorted accordingly. If the sorting is done using sort fields, any `string` fields are compared according to the value of the `caseSensitive` property.

Caution: If you have pending JSDO changes that you need to apply to the server, be sure **not** to invoke this method **before** you invoke the `saveChanges( )` method to successfully apply these changes to the server. Otherwise, the affected client data will be inconsistent with the corresponding data on the server.

Example

The following code fragment shows a JSDO created so it **does not** automatically accept or reject changes to data in JSDO memory after a call to the `saveChanges( )` method. Instead, it subscribes a handler for the JSDO `afterSaveChanges` event to determine if all changes to the `eCustomer` table in JSDO memory should be accepted or rejected based on the success of all resource Create, Update, and Delete operations on the server. To change the data for a record, a jQuery event is also defined on an update button to update the corresponding `eCustomer` record in JSDO memory with the current field values entered in a customer detail form (`#custdetail`):

```
dataSet = new progress.data.JSDO( { name: 'dsCustomerOrder',
                                    autoApplyChanges : false } );
dataSet.eCustomer.subscribe('afterSaveChanges', onAfterSaveCustomers, this);

$('#btnUpdate').bind('click', function(event) {
  var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
  if (jsrecord) {jsrecord.assign();};
});

// Similar controls might be defined to delete and create eCustomer records...

$('#btnSave').bind('click', function(event) {
  dataSet.saveChanges();
});

function onAfterSaveCustomers(jsdo, success, request) {
  if (success)
  {
    jsdo.eCustomer.acceptChanges();
    // Additional actions associated with accepting the pending changes...
  }
  else
  {
    jsdo.eCustomer.rejectChanges();
    // Additional actions associated with rejecting the pending changes...
  }
}
```

When the update button is clicked, the event handler uses the `findById( )` method to find the original record (`jsrecord`) with the matching internal record ID (`#id`) and invokes the `assign( )` method on `jsrecord` with an empty parameter list to update its fields in `eCustomer` with any new values entered into the form. You might define similar events and controls to delete `eCustomer` records and add new `eCustomer` records.

A jQuery event also defines a save button that when clicked invokes the `saveChanges( )` method to apply all pending changes in JSDO memory to the server. After the method completes, and all results have been returned to the client from the server, the JSDO `afterSaveChanges` event fires, and if all resource operations on the server were successful, the handler calls `acceptChanges( )` to accept the pending changes to `eCustomer` in JSDO memory. For more information on how this same example determines when and how to reject changes, see the description of the `rejectChanges( )` method.

Note: This example shows the default invocation of `saveChanges( )`, which invokes each resource record-change operation, one record at a time, across the network. For a resource that supports before-imaging (such as an OpenEdge ProDataSet), you can also have `saveChanges( )` send all pending record change operations across the network in a single Submit operation. For more information and an example, see the description of the `saveChanges( )` method.

See also:

acceptRowChanges( ) method on page 47, autoApplyChanges property on page 82, autoSort property on page 82, caseSensitive property on page 93, rejectChanges( ) method on page 165, saveChanges( ) method on page 174

acceptRowChanges( ) method

Accepts changes to the data in JSDO memory for a specified record object.

This can be the working record of a table reference or the record specified by a `JSRecord` object reference. If the method succeeds, it returns `true`. Otherwise, it returns `false`.

Note: This method is most useful when the JSDO `autoApplyChanges` property is set to `false`. In this case, you might invoke this method in the callback for the corresponding JSDO event fired in response to a successful record-change operation on the server resource that was invoked by executing the `saveChanges( )` method. If the `autoApplyChanges` property is `true`, the JSDO automatically accepts or rejects changes to the record object based on the success of the corresponding resource operation on the server.

Return type: `boolean`

Applies to: progress.data.JSRecord class on page 35, progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Working record: The working record is set depending on the changes accepted.

Syntax

```
jsrecord-ref.acceptRowChanges ( )
jsdo-ref.acceptRowChanges ( )
jsdo-ref.table-ref.acceptRowChanges ( )
```

jsrecord-ref

A reference to a `JSRecord` object for a table reference in JSDO memory.

You can obtain a `JSRecord` object by:

- Invoking a JSDO method that returns record objects from a JSDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a JSDO table reference that already has a working record.
- Accessing the *`record`* parameter passed to the callback of a JSDO `afterCreate`, `afterDelete`, or `afterDelete` event.

jsdo-ref

A reference to the JSDO. You can call the method on *`jsdo-ref`* if the JSDO has only a single table reference, and that table reference has a working record.

table-ref

A table reference on the JSDO that has a working record.

When you accept changes on a specified record object, this method makes the record reflect all pending changes to it in JSDO memory. As the specified changes are accepted, the method also empties any associated before-image data, clears any associated error message setting, and removes the associated pending change indications from JSDO memory.

Note: Regardless if you call `acceptRowChanges( )`, any error message settings that result from Data Object resource operations invoked by the most recent execution of the `fill( )` or `saveChanges( )` method remain available for return by the `getErrors( )` method until the next execution of either `fill( )` or `saveChanges( )`. For more information, see the getErrors( ) method on page 110 description.

Note: After this method accepts changes on a record, and if you have set up automatic sorting using the `autoSort` property, all the record objects for the affected table reference are sorted accordingly. If the sorting is done using sort fields, any `string` fields are compared according to the value of the `caseSensitive` property.

Caution: If you have pending JSDO changes that you need to apply to the server, be sure **not** to invoke this method **before** you invoke the `saveChanges( )` method. Otherwise, the affected client data will be inconsistent with the corresponding data on the server.

Example

The following code fragment shows a JSDO created so it **does not** automatically accept or reject changes to data in JSDO memory after a call to the `saveChanges( )` method. Instead, it subscribes a single handler for each of the `afterDelete`, `afterCreate`, and `afterUpdate`, events to determine if changes to any `eCustomer` table record in JSDO memory should be accepted or rejected based on the success of the corresponding resource operation on the server. To change the data for a record, a jQuery event is also defined on a save button to update the corresponding `eCustomer` record in JSDO memory with the current field values entered in a customer detail form (`#custdetail`):

```
dataSet = new progress.data.JSDO( { name: 'dsCustomerOrder',
                                    autoApplyChanges : false } );
dataSet.eCustomer.subscribe('afterDelete', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterCreate', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterUpdate', onAfterCustomerChange, this);

$('#btnSave').bind('click', function(event) {
  var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
  if (jsrecord) {jsrecord.assign();};
  dataSet.saveChanges();
});

// Similar controls might be defined to delete and create eCustomer records...

function onAfterCustomerChange(jsdo, record, success, request) {
  if (success)  {
    record.acceptRowChanges();
    // Perform other actions associated with accepting this record change
  }
  else
  {
    record.rejectRowChanges();
  }
}
```

When the button is clicked, the event handler uses the `findById( )` method to find the original record with the matching internal record ID (`#id`) and invokes the `assign( )` method on `jsrecord` with an empty parameter list to update its fields in `eCustomer` with any new values entered into the form. It then calls the `saveChanges( )` method to invoke the resource Update operation to apply these record changes to the server. You might define similar events and controls to delete the `eCustomer` record or add a new `eCustomer` record.

After each resource operation for a changed `eCustomer` record completes, results of the operation are returned to the client from the server, and the appropriate event fires. If the operation was successful, the handler calls `acceptRowChanges( )` to accept the record change associated with the event in JSDO memory. An advantage of using an event to manually accept a record change is that you can perform other actions associated with accepting this particular change, such as creating a local log that describes the change.

Note: This example shows the default invocation of `saveChanges( )`, which invokes each resource record-change operation, one record at a time, across the network. For a resource that supports before-imaging (such as an OpenEdge ProDataSet), you can also have `saveChanges( )` send all pending record change operations across the network in a single Submit operation. For more information and an example, see the description of the `saveChanges( )` method.

See also:

acceptChanges( ) method on page 44, autoApplyChanges property on page 82, autoSort property on page 82, caseSensitive property on page 93, rejectRowChanges( ) method on page 168, saveChanges( ) method on page 174

add( ) method

Creates a new record object for a table referenced in JSDO memory and returns a reference to the new record.

To synchronize the change on the server, call the `saveChanges( )` method.

Alias: `create( )`

Return type: progress.data.JSRecord class on page 35

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Working record: After completing execution, the new record becomes the working record for the associated table. If the table has child tables, the working record for these child tables is not set.

Syntax

```
jsdo-ref.add ( [new-record-object] )
jsdo-ref.table-ref.add ( [new-record-object] )
```

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference.

table-ref

A table reference on the JSDO.

new-record-object

If specified as a non-`null` object, passes in the data to create the record for the `JSRecord` instance. The data to create the record is identified by one or more properties, each of which has the name of a corresponding field in the table schema and has the value to set that field in the new table record.

If you omit or set the parameter to `null`, or you do not include properties of *new-record-object* for all fields in the new record, the method uses the default values from the table schema stored in the catalog to set the unspecified record fields.

Note: After this method adds the new record object, and if you have set up automatic sorting using the `autoSort` property, all the record objects for the affected table reference are sorted accordingly. If the sorting is done using sort fields, any `string` fields are compared according to the value of the `caseSensitive` property.

If the specified table reference is for a child table in a multi-table resource, when the `useRelationships` property is `true`, `add( )` uses the relationship to set related field values of the new child record from the working record of the parent table. However, if the working record of the parent is not set, `add( )` throws an error. If `useRelationships` is `false`, the fields for the new record are set as specified by *new-record-object* and no error is thrown.

Example

Assuming `useRelationships` is `true`, given a JSDO created for a multi-table resource with a `customer` and related child `order` table, the `add( )` method in the following code fragment uses this relationship to automatically set the `CustNum` field in a new record added to the `order` table:

```
var dataSet = new Progress.data.JSDO( 'CustomerOrderDS' );
dataSet.customer.add( { CustNum: 1000, Balance: 10000, State: 'MA' } );

// CustNum is set automatically by using the relationship
dataSet.order.add( { OrderNum: 1000 } );
```

See also:

autoSort property on page 82, caseSensitive property on page 93, fill( ) method on page 97, getId( ) method on page 116, getSchema( ) method on page 120, data property on page 96, saveChanges( ) method on page 174

addCatalog( ) method (JSDOSession class)

Loads one or more local or remote Data Service Catalogs into the current `JSDOSession` object.

The appropriate Catalog must be loaded before creating a JSDO for any resource defined in the Catalog.

If a Catalog is on a web server (remote), this method throws an exception if it is not possible to send a request to the specified web application.

This method is always executed asynchronously and returns results in callbacks that you register using methods of a Promise object returned as the method value.

Note: In order to invoke this method successfully, jQuery Promises must be supported in your development environment. Otherwise, the method throws an exception.

Note: You can call this method either before or after the `JSDOSession` object has successfully established a JSDO login session using its `login( )` method. However, if the Data Service Catalog is stored remotely in the web application that hosts the Data Object Service, Progress recommends that you log into the web application **before** calling this method.

Note: As a recommended alternative to using this method on an existing `JSDOSession` object, you can create a new, initialized `JSDOSession` using the `progress.data.getSession( )` stand-alone function. In one call, this function instantiates the object, invokes its `login( )` method with specified credentials, then invokes its `addCatalog( )` method to load a specified Data Service Catalog. For more information, see the getSession( ) stand-alone function on page 121 description.

Return type: jQuery Promise

Applies to: progress.data.JSDOSession class on page 25

Syntax

```
addCatalog ( catalog-uri [ , cat-user-name , cat-password ]
             [ , parameter-object ] )
addCatalog ( catalog-uris [ , cat-user-name , cat-password ]
             [ , parameter-object ] )
```

catalog-uri

A `string` expression that specifies the URI of a Data Service Catalog file. This URI can specify either a location (remote) in a web application running on a web server or a location (local) that is relative to the device where the app is currently running. If the URI specifies a remote location, it is typically the location of the web application that hosts the Data Object Service and where the `JSDOSession` object has a JSDO login session. If the URI is a relative path (e.g., `catalogs/OrderEntrySvc.json`), the *`catalog-uri`* is assumed to be relative to the location from which the app was loaded.

If the mobile app from which you are logging in is a hybrid app that will be installed to run directly in a native device container, or if it is a web app deployed to a different web server from the web application to which the `JSDOSession` object has already logged in, you must specify an absolute URI for the Catalog that includes the Tomcat server domain or host and port, for example, `http://www.progress.com:8980/SportsMobileApp/static/OrderEntrySvc.json`, or perhaps for testing, `http://localhost:8980/SportsMobileApp/static/OrderEntrySvc.json`.

If the `JSDOSession` object has already logged in and the mobile app is a web app deployed to the same Apache Tomcat server as the web application that hosts the Data Object Service, you can specify a URI that is relative to the deployment end point (Tomcat server domain or host and port), for example: `/SportsMobileApp/static/OrderEntrySvc.json`, where `/SportsMobileApp` is the location of the web application.

Note: The default Catalog URI for a Catalog created for an OpenEdge Data Object Service, relative to the Apache Tomcat server domain or host and port where the session is logged in, is the following: */`WebApplicationName`*`/static/`*`ServiceName`*`.json`, where *`WebApplicationName`* is the name of the web application and *`ServiceName`* is the name of the Data Object Service for which the Data Service Catalog is created.

catalog-uris

An array of `string` expressions, each of which is the URI for a Catalog as defined for the *`catalogURI`* parameter. If any of the Catalogs in the array are protected separately from the web application where the `JSDOSession` object is logged in, you can specify the additional credentials using *`cat-user-name`* and *`cat-password`*.

Note: You can read the `catalogURIs` property to return the URIs for all Catalogs previously loaded into the `JSDOSession` object.

cat-user-name

A `string` expression containing a user ID to authenticate access to a protected Catalog. If you do not specify *`cat-user-name`*, Catalog access is authorized using existing user credentials (if any).

cat-password

A `string` expression containing a password (if required) to authenticate the user specified by *`cat-user-name`*.

parameter-object

An object that contains the following optional property:

- **`iOSBasicAuthTimeout`** — A `number` that specifies the time, in milliseconds, that the `addCatalog( )` method waits for a response before returning an error. This error might mean that the user entered invalid credentials to access a protected Catalog URI. If you set this value to zero (0), no timeout is set, and `addCatalog( )` can wait indefinitely before returning an error. If you do not set the `iOSBasicAuthTimeout` property, `addCatalog( )` uses 4000 (4 seconds) as the default value.

  Note: Any non-zero timeout value (default or otherwise) for this *`parameter-object`* property operates **only** under certain conditions. Otherwise, any setting of this property has no effect. For more information, see the notes of the progress.data.JSDOSession class on page 25 description.

Note: If the `JSDOSession` object is already logged in, you typically do not need to specify *`cat-user-name`* and *`cat-password`*. These optional parameters are available primarily if the object is not yet logged in, or if you store the Catalog someplace on the web other than the web application that hosts the Data Object Service (where the `JSDOSession` object has a JSDO login session).

Caution: Although you can use a single `JSDOSession` object to load a Catalog that is stored somewhere on the web other than the web application that hosts the Data Object Service, this typically works only if the Catalog is protected with Anonymous or HTTP Basic authentication.

Promise method signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that `addCatalog( )` returns:

Syntax:

```
promise.done( function ( session , result , details ) )
promise.fail( function ( session , result , details ) )
promise.always( function ( session , result , details ) )
```

promise

A reference to the Promise object that is returned as the value of the `addCatalog( )` method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDOSession class on page 25.

session

A reference to the `JSDOSession` object on which `addCatalog( )` was called.

result

A constant indicating the overall result of the call that can have one of the following values:

- **`progress.data.Session.SUCCESS`** — Each Catalog specified by the *`catalogURI`* or *`catalogURIs`* parameter of `addCatalog( )` has been loaded successfully or has already been loaded.
- **`progress.data.Session.GENERAL_FAILURE`** — One or more of the specified Catalogs has failed to load successfully.

Note: It is not always necessary to test the value of *`result`* in a Promise method callback for the `addCatalog( )` method, especially if the callback is registered using *`promise`*`.done( )` and *`promise`*`.fail( )`, where each callback always executes with the same value for success (`done( )`) or failure (`fail( )`).

---

details

An array of JavaScript objects that contain information on the Data Service Catalogs that `addCatalog( )` attempted to load. Each object has the following properties:

- **`catalogURI`** — The URI of a specified Catalog.
- **`result`** — A constant indicating the result of loading the Catalog that can have one of the following values:
  - **`progress.data.Session.SUCCESS`** — The specified Catalog loaded successfully, or has previously been loaded.
  - **`progress.data.Session.CATALOG_ALREADY_LOADED`** — The specified Catalog was previously loaded.

    Note: When `addCatalog( )` is requested to load a Data Service Catalog that is already loaded, it does not load the Catalog again, but is considered a successful execution. That is, the *`promise`*`.done( )` callback is called, although the `result` value in the *`details`* object for the specified Catalog is `progress.data.Session.CATALOG_ALREADY_LOADED`.

  - **`progress.data.Session.AUTHENTICATION_FAILURE`** — The specified Catalog failed to load because of an authentication error.

- **`progress.data.Session.GENERAL_FAILURE`** — The specified Catalog failed to load because of an error other than an authentication failure.

  Note: This value can also be returned if invalid user credentials triggered a Catalog access timeout according to the value of the *`parameter-object`*`.iOSBasicAuthTimeout` property.

- **`errorObject`** — Any error object thrown while attempting to load the Catalog.

  Note: If this object is thrown because of a Catalog access timeout triggered according to the value of the *`parameter-object`*`.iOSBasicAuthTimeout` property, the `message` property of the error object is set to `"addCatalog timeout expired"`.

- **`xhr`** — A reference to the XMLHttpRequest object used to load the Catalog from a web server.

Example

The following code fragment calls the `addCatalog(myCatalogURIs)` method on the `JSDOSession` object, `mySession`, to load multiple Data Service Catalogs specified in `myCatalogURIs`. This example uses `try` and `catch` blocks to check any error object thrown prior to requesting the Catalog, and assembles an appropriate message to display in an alert box or does other processing for each case:

```
try {
    mySession.addCatalog( myCatalogURIs ).done(
        function( session, result, details ) {
            alert("All Catalogs loaded.");
        }).fail(
        function( session, result, details ) {
            var numCats = details.length;
            for ( i = 0; i < numCats; i++ ) {
                if (details[i].result
                    === progress.data.Session.AUTHENTICATION_FAILURE) {
                    alert("Authentication error: " + details[i].catalogURI);
                } else if (details[i].result
                           === progress.data.Session.GENERAL_FAILURE) {
                    alert("General Catalog load error: "
                          + details[i].catalogURI);
                    if (details[i].errorObject) {
                        // Process thrown error object during load . . .
                    }
                    if (details[i].xhr) {
                        // Process XHR object sent for the load . . .
                    }
                } else {
                    alert("Not sure what is wrong with "
                          + details[i].catalogURI);
                }
            } // for each load attempt
        });
}
catch(errObj) {
    var msg;

    msg = ‘\n’ + errObj.message;
    alert("Unexpected addCatalog() error: " + msg);
}
```

See also:

catalogURIs property on page 94, getSession( ) stand-alone function on page 121, login( ) method (JSDOSession class) on page 145

addLocalRecords( ) method

Reads the record objects stored in the specified local storage area and updates JSDO memory based on these record objects, including any pending changes and before-image data, if they exist.

This method updates all affected tables of the JSDO with the locally stored record objects. These record objects are merged into JSDO memory and affect existing data according to a specified merge mode and optional key fields.

Return type: `boolean`

Applies to: progress.data.JSDO class on page 12

Working record: After execution, the working record set for each table in the JSDO depends on the specified merge mode.

Syntax

```
addLocalRecords ( [ storage-name , ] add-mode [ , key-fields ] )
```

storage-name

The name of the local storage area from which to update JSDO memory. If *storage-name* is not specified, blank, or `null`, the name of the default storage area is used. The name of this default area is `jsdo_`*`serviceName`*`_`*`resourceName`*, where *serviceName* is the name of the Data Object Service that supports the JSDO instance, and *resourceName* is the name of the resource (table, dataset, etc.) for which the JSDO instance is created.

Note: Record objects read in from local storage can contain before-image data, which this method merges into JSDO memory along with the data from the record objects. However, if a record object read in from local storage contains before-image data that conflicts with existing before-image data in JSDO memory for that same record object, `addLocalRecords( )` throws an exception.

---

add-mode

An integer constant that represents a merge mode to use. Each merge mode handles duplicate keys in a particular manner, depending on your specification of *key-fields*. You can specify the following numeric constants, which affect how the table record objects in the specified local storage area are added to JSDO memory:

- **`progress.data.JSDO.MODE_REPLACE`** — Adds the table record objects in the specified local storage area to the existing record objects in JSDO memory. If duplicate keys are found between record objects in local storage and record objects in JSDO memory, the record objects with duplicate keys in JSDO memory are replaced with the corresponding records in local storage.

Caution:

If any specified *key-fields* match the unique indexes of corresponding tables on the server, adding the contents of the specified local storage area can result in records with duplicate keys. If the corresponding server tables have unique indexes, you must make any affected duplicate key fields unique before calling `saveChanges( )`.

key-fields

An object with a list of primary key fields to check for records with duplicate keys. For example, when merging with a multi-table resource that has `eCustomer` and `eOrder` table references, you might use the following object:

```
{
  eCustomer: [ "CustNum" ],
  eOrder: [ "CustNum", "Ordernum" ]
}
```

When merging with a single-table reference, you might use the following array object:

```
[ "CustNum", "Ordernum" ]
```

Note: For any *key-fields* that have the `string` data type, the character values for these fields are compared to identify duplicates according to the value of the `caseSensitive` property on each affected table reference.

If *key-fields* is specified, the method checks for duplicate keys using the specified primary keys found in *key-fields*. If *key-fields* is **not** specified, the method searches other possible sources for definitions of primary keys in the following order, and uses the first source of definitions found:

1. Primary key annotations from any OpenEdge resource (as identified in the Data Service Catalog)
2. Unique ID properties associated with the resource (for example, the `idProperty` property as identified in the Data Service Catalog)

If no source of primary key definitions is found, the method adds **all** local storage records to JSDO memory, regardless of the specified *add-mode*, and regardless of any duplicate records that might result.

Note: After this method checks for any duplicate keys and completes adding record objects to JSDO memory, and if you have set up automatic sorting using the `autoSort` property, all the record objects for the affected table references are sorted accordingly. If the sorting is done using sort fields, any `string` values in the specified sort fields are compared according to the value of the `caseSensitive` property.

This method returns `true` if it successfully reads the data from the local storage area; it then updates JSDO memory with this data according to the specified *add-mode*. If *storage-name* does not exist, but otherwise encounters no errors, the method leaves JSDO memory unchanged and returns `false`. If the method does encounter errors (for example, with reading the data in the specified storage area), it also leaves JSDO memory unchanged and throws an exception.

Example

The following code fragment fills memory for a JSDO (`dataset`) with records from an OpenEdge ProDataSet named `csCustomerOrder`, which is used to implement a multi-table resource on the server. The JSDO then contains tables that correspond to the `Customer` and `Order` tables of the OpenEdge `sports2000` database:

```
var dataset = progress.data.JSDO( "dsCustomerOrder" );
dataset.fill(); // Loads the JSDO with all available records from the ProDataSet resource

// Adds records
dataset.addLocalRecords( progress.data.JSDO.MODE_REPLACE, [ "CustNum", "Ordernum" ] );
```

The fragment then calls `addLocalRecords( )` on the JSDO to add a set of similar records to JSDO memory from the default local storage area, where the records were previously stored using the JSDO `saveLocal( )` method. Duplicate `Customer` and `Order` records are checked and replaced with the records from local storage based on the respective primary key fields, `CustNum` and `Ordernum`.

See also:

autoSort property on page 82, caseSensitive property on page 93, fill( ) method on page 97, getId( ) method on page 116, readLocal( ) method on page 164, saveChanges( ) method on page 174, saveLocal( ) method on page 188

addPlugin( ) method

Adds (registers) a JSDO plugin object with a specified name in the current document context that can be used with a supported Data Object operation invoked on a JSDO instance.

Return type: `null`

Applies to: progress.data.PluginManager class on page 36

Syntax

```
progress.data.PluginManager.addPlugin( name , plugin )
```

name

A `string` expression that evaluates to the name of the plugin. If a plugin by this name already exists, the method throws an exception.

Note: The `PluginManager` class provides access to a built-in plugin with the reserved name, `"JFP"`, which you can customize as described further in this topic. You can also customize any other JSDO plugin that you have already registered.

---

plugin

A plugin `Object` that you can define with certain properties. Each property references the definition of a function that you provide. Each function executes according to the requirements of the supported Data Object operation invoked by any JSDO instance.

This method currently supports only MappingType plugins registered for the Read operation. A MappingType plugin object can contain any or all of the following properties:

- **`requestMapping`** — Defines a function that **must** return an object with properties that define the parameters passed to the JSDO Read operation method (`fill( )` or `read( )`). This function thus allows you to specify or modify any or all of the properties that you can include in a Read operation method's *`parameter-object`* argument before the method executes, thus affecting how the Read operation request is invoked across the network. The function must have the following signature:

```
function ( jsdo , params , info )
```

`jsdo`

A reference to the `progress.data.JSDO` object that corresponds to the JSDO on which the Read operation method was invoked.

`params`

A reference to the `Object` passed to the Read operation method as its *`parameter-object`* argument.

`info`

A reference to an `Object` with a single `operation` property set to the value, `"read"`.

- **`responseMapping`** — Defines a function that **must** return an object with properties that provide the data returned from the corresponding Read operation executed on the server. This function allows you to process the network response from the Read operation in some way. The function must have the following signature:

```
function ( jsdo , response , info )
```

`jsdo`

A reference to the `progress.data.JSDO` object that corresponds to the JSDO on which the Read operation method was invoked.

`response`

A reference to the `Object` that contains data properties returned from the Read operation across the network. For more information, see the description of the response property on page 172 of JSDO request object for returning Read operation output.

Note: If this `responseMapping` function customizes the built-in `"JFP"` plugin or defines a custom plugin based on this built-in plugin, in the body of the function, you can set the value of the JSDO user-defined `"server.count"` property to a corresponding *`response`* property that holds the total count of records returned in the Read operation result set. This has the equivalent effect of specifying a Count operation for the server Data Object to return the same value. For more information, see the Example on page 59 in this topic.

`info`

A reference to an `Object` with a single `operation` property set to the value, `"read"`.

The built-in "JFP" MappingType plugin is available to support JSDO access by a Kendo UI DataSource. For more information on this "JFP" plugin, see the description of the fill( ) method on page 97. Although you cannot create your own plugins with this name, you can reference the built-in "JFP" plugin, for example, to create a custom requestMapping or responseMapping function for it, or to create an entirely new plugin based on the existing built-in "JFP" plugin. For more information see the Example on page 59 in this topic.

To allow an OpenEdge Data Object Read operation to use either the built-in "JFP" or your custom MappingType plugin, you must annotate the ABL method that implements the operation to identify the plugin to use. For more information, see the Notes on page 60 in this topic.

Example

The following JSDO plugin example adds a new plugin, "MYJFP", which is based on the built-in "JFP" plugin. The example begins by returning a reference to the built-in "JFP" plugin object in the current document context as jfpPlugin and invokes addPlugin( ) to register the new "MYJFP" plugin, as follows:

```
jsdo = new progress.data.JSDO( . . . );

var jfpPlugin = progress.data.PluginManager.getPlugin("JFP");

progress.data.PluginManager.addPlugin("MYJFP", {

    requestMapping: function(jsdo, params, info) {

        var requestParams = {},
            object = {};
        params = jfpPlugin.requestMapping(jsdo, params, info);
        if (params && typeof params.filter === "string") {
            object = JSON.parse(params.filter);
        }
        object.mydata = jsdo.getProperty("mydata");
        requestParams.filter = JSON.stringify(object);

        // You must return the [updated] parameter object
        return requestParams;

    }

    responseMapping: function(jsdo, response, info) {

        var record;
        var newData = response.dsEmployee.ttEmployee;
        if (info.operation === "read") {

            for (var i = 0; i < newData.length; i++) {
                record = newData[i];
                record.VacDays = record.VacDays + 10;
            }

            jsdo.setProperty("server.count", response.myTotal);
        }

        // You must return the [updated] response object
        return response;

    }
});
```

For the new requestMapping function, the example builds and returns a custom params object (requestParams) from both the params object returned by the existing requestMapping function invoked on the built-in "JFP" plugin and an additional mydata property returned as a custom JSDO property that has already been set on the currently executing JSDO instance (jsdo).

For the new `responseMapping` function, the example first tests that the function is executing in the context of a Read operation and applies a common update to the returned `dsEmployee.ttEmployee` table, adding 10 days to the `VacDays` field of each record. It then sets the reserved custom JSDO property, `"server.count"`, on the currently executing JSDO instance (`jsdo`) to the custom value returned from the server as `response.myTotal` before returning the updated `response` from the Read operation.

Notes

- If you want to dynamically update the definition for an existing plugin, whether it is the built-in `"JFP"` plugin or a custom plugin you have previously registered, you can invoke the `getPlugin( )` method to return a reference to the plugin object and assign the appropriate property to the updated function definition. For example, to assign an updated definition for the `responseMapping` function of an existing custom plugin named `"myReadPlugin"`, you can invoke code similar this:

```
progress.data.PluginManager.getPlugin("myReadPlugin").responseMapping
    = function(jsdo, response, info) { /* Your new definition */ };
```

  For an example, see the description of the getPlugin( ) method on page 117.

- For an OpenEdge Data Object Read operation to use the specified MappingType plugin from the client document context, you must properly annotate the definition of the Read operation method in the ABL Business Entity that defines the server Data Object. This annotation defines a `mappingType` property in the Data Service Catalog with a string value that is identical to the name of a MappingType plugin you have registered for the JSDO instance on the client. You specify this annotation with the following syntax:

```
@openapi.openedge.method.property (name="mappingType", value="plugin-name").
```

  Where "*plugin-name*" is the quoted name that you used to register the custom plugin with the `addPlugin( )` method. For example, on the definition of an ABL Read operation method, `ReadEmployee( )`, you might add this annotation for a plugin registered with the name `"MYJFP"`:

```
@openapi.openedge.export(type="REST", useReturnValue="false",
writeDataSetBeforeImage="true").
@progress.service.resourceMapping(type="REST", operation="read",
URI="?filter=~{filter~}", alias="", mediaType="application/json").
@openapi.openedge.method.property (name="mappingType", value="MYJFP").
METHOD PUBLIC VOID ReadEmployee(
    INPUT filter AS CHARACTER,
    OUTPUT DATASET dsEmployee):
    SUPER:ReadData(filter).

END METHOD.
```

  When this `ReadEmployee( )` method executes on the server, it uses the parameter values passed by the client JSDO and provided by any `requestMapping` function defined in the `"MYJFP"` JSDO plugin, and the JSDO then handles the response according to the processing provided by any `responseMapping` function defined in the same plugin.

Note: When annotating for use of the built-in `"JFP"` plugin only, you must also add an annotation that defines an associated `capabilities` property for that plugin. For more information on annotations required for using the built-in `"JFP"` plugin, see the description of Business Entity updates for access by the Kendo UI DataSource in *OpenEdge Development: Web Services*. However note, if you are annotating a custom plugin that is **based** on the built-in `"JFP"` plugin (such as the sample `"MYJFP"` plugin annotated here), you do **not** also need to annotate an associated `capabilities` property for that custom plugin.

See also

fill( ) method on page 97, getPlugin( ) method on page 117, progress.data.JSDO class on page 12

addRecords( ) method

Updates JSDO memory with one or more record objects read from an array, single-table, or multi-table resource that are passed in an object parameter, including any pending changes and before-image data, if they exist.

This method updates all tables or a specified table in JSDO memory, depending on how the method is called. The data is merged into JSDO memory and affects existing data according to a specified merge mode and optional key fields.

Return type: `null`

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Working record: After execution, the working record set for each table in the JSDO depends on the specified merge mode.

Syntax

```
jsdo-ref.addRecords ( merge-object , add-mode [ , key-fields ] )
jsdo-ref.table-ref.addRecords ( merge-object , add-mode [ , key-fields ] )
```

jsdo-ref

A reference to the JSDO. If you call the method on *jsdo-ref*, the method merges data for all referenced tables in JSDO memory.

table-ref

A table reference on the JSDO. If you call the method on *table-ref*, the method merges data only for the referenced table in JSDO memory.

merge-object

An object with the data to merge. If you call the method on *table-ref*, the object can either be an object that contains an array of record objects to merge with the referenced table or a multi-table-formatted object containing such an array.

Note: This object must have a supported JavaScript object format that matches the data returned from the resource Read operation (JSDO `fill( )` method). For example, an object returned as output from an invocation method called on a single-table or multi-table resource will work if it has the same schema as output from the resource Read operation.

The following formats are supported for *`merge-object`*:

- A single table object with an array of record objects. For example:

```
{
  eCustomer: [
    // Record objects ...
  ]
}
```

- An array of record objects for either a single table object or a multi-table object containing only a single table object. For example:

```
[
  // Record objects ...
]
```

- A multi-table object with a single table object or multiple table objects at the same level only. For example:

```
{
  dsCustomerOrder: {
    eCustomer: [
      // Record objects ...
    ],
    eOrder: [
      // Record objects ...
    ]
  }
}
```

Note: The record objects passed in *`merge-object`* can contain before-image data, which this method merges into JSDO memory along with the data from the record objects. However, if a record object read in from local storage contains before-image data that conflicts with existing before-image data in JSDO memory for that same record object, `addLocalRecords( )` throws an exception.

`add-mode`

An integer that represents a merge mode to use. If you also specify *`key-fields`*, each merge mode handles duplicate keys in a particular manner as described here. If you **do not** specify *`key-fields`*, the method adds **all** the records of *`merge-object`* regardless of the mode. You can specify the following numeric constants, which affect how the table record objects in *`merge-object`* are added to JSDO memory:

- **progress.data.JSDO.MODE_APPEND** — Adds the table record objects in *merge-object* to the existing record objects in JSDO memory. If a duplicate key is found between a record object in *merge-object* and a record object in JSDO memory, the method throws an error.
- **progress.data.JSDO.MODE_MERGE** — Adds the table record objects in *merge-object* to the existing record objects in JSDO memory. If duplicate keys are found between record objects in *merge-object* and record objects in JSDO memory, the method ignores (does not add) the record objects with duplicate keys in *merge-object*.
- **progress.data.JSDO.MODE_REPLACE** — Adds the table record objects in *merge-object* to the existing record objects in JSDO memory. If duplicate keys are found between record objects in *merge-object* and record objects in JSDO memory, the record objects with duplicate keys in JSDO memory are replaced with the corresponding records in *merge-object*.
- **progress.data.JSDO.MODE_EMPTY** — Empties all table record objects from JSDO memory and replaces them with the contents of *merge-object*.

Note: If *merge-object* is an empty object ({}), this mode effectively empties the data from JSDO memory.

After execution, if the specified merge mode was progress.data.JSDO.MODE_EMPTY, the working record set for any table references is undefined, because JSDO memory is completely emptied or replaced. For any other merge mode, the working record set for each JSDO table reference remains unchanged.

Caution:

If a table *key-fields* matches the unique indexes of corresponding tables, adding the contents of *merge-object* can result in records with duplicate keys. If the corresponding tables have unique indexes, you must make any affected duplicate key fields unique before calling saveChanges( ).

key-fields

An object with a list of key fields to check for records with duplicate keys. For example, when merging into a JSDO that has eCustomer and eOrder table references, you might use the following object:

```
{
  eCustomer: [ "CustNum" ],
  eOrder: [ "CustNum", "Ordernum" ]
}
```

When merging with a single table reference, you might use the following array object:

```
[ "CustNum", "Ordernum" ]
```

Note: For any *key-fields* that have the string data type, the character values for these fields are compared to identify duplicates according to the value of the caseSensitive property on each affected table reference.

Note: After this method checks for any duplicate keys and completes adding record objects to JSDO memory from *merge-object*, and if you have set up automatic sorting using the autoSort property, all the record objects for the affected table references are sorted accordingly. If the sorting is done using sort fields, any string values in the specified sort fields are compared according to the value of the caseSensitive property.

A typical use for addRecords( ) is to merge additional data returned by an invocation method without having to re-load JSDO memory with all the data from the fill( ) method.

Example

Given a JSDO (dataset) that you fill with available records from the eCustomer and eOrder tables of an OpenEdge ProDataSet, you might retrieve a new eOrder record as the result of a getNewOrder( ) invocation method on the JSDO and add the new record to JSDO memory as follows:

```
var dataset = progress.data.JSDO( "dsCustomerOrder" );
dataset.fill(); // Loads the JSDO with all available records from the ProDataSet resource

// Adds a new eOrder record restrieved from the service
var request = dataset.getNewOrder(null,false);
dataset.eOrder.addRecords( request.response, progress.data.JSDO.MODE_APPEND,
  [ "CustNum", "Ordernum" ],
  );
```

This code fragment adds the eOrder record for an existing eCustomer record specified by the CustNum property and a new order number specified by the Ordernum property of the single record object returned in result.dsCustomerOrder.eOrder[0].

See also:

autoSort property on page 82, caseSensitive property on page 93, getId( ) method on page 116, fill( ) method on page 97, invocation method on page 134, saveChanges( ) method on page 174

afterCreate event

Fires after the JSDO, by means of a saveChanges( ) call following an add( ) call, sends a request to create a record in the Data Object resource and receives a response to this request from the server.

This event fires after the response from a Create operation request (sent without using Submit) has been received, and fires after the response from a Submit operation request for each one of possibly multiple record creates has been received.

Note: A single Create operation request is sent for each record object that you have created in the JSDO with a single call to saveChanges( ) or saveChanges(false). A single Submit operation request for any and all created, updated, and deleted JSDO record objects is sent with a single call to saveChanges(true).

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , record , success , request )
```

jsdo

A reference to the JSDO that invoked the create request. For more information, see the description of jsdo property on page 143 of the request object.

record

A reference to the table record upon which the create request acted. For more information, see the description of jsrecord property on page 143 of the request object.

success

A `boolean` that is `true` if the create request was successful. For more information, see the description of success property on page 202 of the request object.

request

A reference to the request object returned after the create request completes. For more information, see the description of request object on page 37.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance or one of its table references.

Example

The following code fragment subscribes the function, `onAfterCreate`, to handle the `afterCreate` event fired on the JSDO, `myjsdo`, created for an OpenEdge single-table resource, `ttCustomer`, where `newRecordData` is an object containing the field values to set in the new record:

```
/* subscribe to event */
myjsdo.subscribe( 'afterCreate', onAfterCreate );

/* some code that might add one or more
   records and save them on the server */
var jsrecord = myjsdo.add( newRecordData );
. . .

myjsdo.saveChanges();

function onAfterCreate ( jsdo , record , success , request ) {
    var myField,
        lenErrors,
        errors,
        errorType;
    if (success) {
        /* for example, get the values from the record for redisplay */
        myField = record.data.myField;
        . . .
    }
    else { /* Handle Create operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "BI Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
            }
            /* Log error type and message for current error */
            console.log("ERROR: " + errorType + errors[idxError].error);
            if (errors[idxError].id) { /* Log info for error with record object */
                console.log("RECORD ID: " + errors[idxError].id);
                /* If the error record ID matches the operation
                   record ID, log the record data values */
                if  (errors[idxError].id == record.getId()) {
                    console.log ("Customer Record Fields:");
                    console log ("    CustNum = " + record.data.CustNum);
                    . . .
                }
            }
            /* Log additional HTTP text for web or other server errors */
            if (errors[idxError].responseText) {
                console.log("HTTP FULL TEXT: " + errors[idxError].responseText);
            }
        }
    } /* End Create operation errors */
};
```

See also:

add( ) method on page 49, beforeCreate event on page 85, record property on page 165, saveChanges( ) method on page 174, subscribe( ) method (JSDO class) on page 200, unsubscribe( ) method (JSDO class) on page 206

afterDelete event

Fires after the JSDO, by means of a `saveChanges( )` call following a `remove( )` call, sends a request to delete a record in the Data Object resource and receives a response to this request from the server.

This event fires after the response from a Delete operation request (sent without using Submit) has been received, and fires after the response from a Submit operation request for each one of possibly multiple record deletes has been received.

Note: A single Delete operation request is sent for each record object that you have deleted in the JSDO with a single call to `saveChanges( )` or `saveChanges(false)`. A single Submit operation request for any and all created, updated, and deleted JSDO record objects is sent with a single call to `saveChanges(true)`.

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , record , success , request )
```

jsdo

A reference to the JSDO that invoked the delete request. For more information, see the description of jsdo property on page 143 of the request object.

record

A reference to the table record upon which the delete request acted. For more information, see the description of jsrecord property on page 143 of the request object.

success

A `boolean` that is `true` if the delete request was successful. For more information, see the description of success property on page 202 of the request object.

request

A reference to the request object returned after the delete request completes. For more information, see the description of request object on page 37.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance or one of its table references.

Example

The following code fragment subscribes the function, `onAfterDelete`, to handle the `afterDelete` event fired on the JSDO, `myjsdo`, created for a single-table resource, `ttCustomer`, where `myid` is the known ID of a record to find and delete:

```
/* subscribe to event */
myjsdo.subscribe( 'afterDelete', onAfterDelete );

/* some code that might delete one or more
   records, then remove them on the server */
var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.remove();};
. . .

myjsdo.saveChanges();

function onAfterDelete ( jsdo , record , success , request ) {
    var myField,
        lenErrors,
        errors,
        errorType;
    if (success) {
        /* for example, get the values from the record that was
           deleted to display a confirmation message */
        myKeyField = record.data.myKeyField;
        . . .
    }
    else { /* Handle Delete operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "BI Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
            }
            /* Log error type and message for current error */
            console.log("ERROR: " + errorType + errors[idxError].error);
            if (errors[idxError].id) { /* Log info for error with record object */
                console.log("RECORD ID: " + errors[idxError].id);
                /* If the error record ID matches the operation
                   record ID, log the record data values */
                if  (errors[idxError].id == record.getId()) {
                    console.log ("Customer Record Fields:");
                    console log ("    CustNum = " + record.data.CustNum);
                    . . .
                }
            }
            /* Log additional HTTP text for web or other server errors */
            if (errors[idxError].responseText) {
                console.log("HTTP FULL TEXT: " + errors[idxError].responseText);
            }
        }
    } /* End Delete operation errors */
};
```

See also:

beforeDelete event on page 87, remove( ) method on page 171, saveChanges( ) method on page 174, subscribe( ) method (JSDO class) on page 200, unsubscribe( ) method (JSDO class) on page 206

afterFill event

Fires on the JSDO after a call to the `fill( )` method executes and returns the results from the server to JSDO memory for a Read operation request on its Data Object resource.

Alias: `afterRead`

Applies to: progress.data.JSDO class on page 12

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , success , request )
```

jsdo

A reference to the JSDO that invoked the `fill( )` method to initialize JSDO memory with the data in its resource. For more information, see the description of the jsdo property on page 143 of the request object.

success

A `boolean` that is `true` if the Read operation on the resource was successful. For more information, see the description of the success property on page 202 of the request object.

request

A reference to the request object returned after the `fill( )` method completed execution either successfully or unsuccessfully. For more information, see the description of the request object on page 37.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance.

Example

The following code fragment subscribes the callback function, `onAfterFill`, to handle the `afterFill` event fired on the JSDO, `myjsdo`, after results are returned for a Read operation on a single-table resource, `ttCustomer`:

```
* subscribe to event */
myjsdo.subscribe( 'afterFill', onAfterFill );

myjsdo.fill();

function onAfterFill( jsdo , success , request ) {
    var lenErrors,
        errors,
        errorType;
    if (success) {
        /* for example, add code to display all records on a list */
        jsdo.foreach(function (jsrecord) {
            /* you can reference the fields as jsrecord.data.field */
        });
    }
    else {/* Handle Read operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
            }
            /* Log error type and message for current error */
            console.log("READ ERROR: " + errorType + errors[idxError].error);
            /* Log additional HTTP text for web or other server errors */
            if (errors[idxError].responseText) {
                console.log("HTTP FULL TEXT: " + errors[idxError].responseText);
            }
        }
    }

};
```

See also:

beforeFill event on page 88, fill( ) method on page 97, subscribe( ) method (JSDO class) on page 200, unsubscribe( ) method (JSDO class) on page 206

afterInvoke event

Fires after a custom invocation method is called asynchronously on a JSDO and a response to the Invoke operation request is received from the server.

Synchronous invocation method calls do not cause this event to fire.

Applies to: progress.data.JSDO class on page 12

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , success , request )
```

jsdo

A reference to the JSDO that invoked the method. For more information, see the description of jsdo property on page 143 of the request object.

success

A `boolean` that is `true` if the operation was successful. For more information, see the description of success property on page 202 of the request object.

request

A reference to the request object returned after the operation completes. For more information, see the description of request object on page 37.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance.

Note: To subscribe a handler for this event, the `subscribe( )` method requires that you pass, as a parameter, the name of the invocation method to which the event applies.

Example

The following code fragment subscribes the function, `onAfterInvokeGetOrderTotalAndStatus`, to handle the `afterInvoke` event fired on the JSDO, `dataSet`, for an invocation of the `getOrderTotalAndStatus( )` invocation method passed the parameters specified by `paramObject`:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );
                  dataSet.subscribe( 'afterInvoke', 'getOrderTotalAndStatus',
                  onAfterInvokeGetOrderTotalAndStatus );

                  dataSet.getOrderTotalAndStatus( paramObject );

                function onAfterInvokeGetOrderTotalAndStatus( jsdo , success , request
 )
                     if (success) {

                         var response = request.response;
                         var ordTotal = response._retVal;
                         var ordStatus = response.pcStatus;
                  /* process successful results */
                  . . .

                     }
                     else {
                         if (request.response && request.response._errors &&
                             request.response._errors.length > 0) {

                             var lenErrors = request.response._errors.length;
                             for (var idxError=0; idxError < lenErrors; idxError++) {


                                 var errorEntry = request.response._errors[idxError];

                                 var errorMsg = errorEntry._errorMsg;
                                 var errorNum = errorEntry._errorNum;
                                 /* handle error */
                  . . .

                             }
                         }
                     }

                  };
```

See also:

beforeInvoke event on page 89, invocation method on page 134, subscribe( ) method (JSDO class) on page 200, unsubscribe( ) method (JSDO class) on page 206

afterSaveChanges event

Fires once for each call to the `saveChanges( )` method on a JSDO, after responses to all create, update, and delete record requests sent to a Data Object resource have been received from the server.

This event fires after all responses have been received from one or more Create, Update, and Delete (CUD) operation requests (sent without using Submit), and fires after the one response has been received from a single Submit operation request after its one or more record changes have completed.

Note: A single CUD operation request is sent for each record object that you have created, updated, or deleted (respectively) in the JSDO with a single call to `saveChanges( )` or `saveChanges(false)`. A single Submit operation request for any and all created, updated, and deleted JSDO record objects is sent with a single call to `saveChanges(true)`.

Applies to: progress.data.JSDO class on page 12

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , success , request )
```

jsdo

A reference to the JSDO that invoked the `saveChanges( )` method for one or more record changes on its resource. For more information, see the description of the jsdo property on page 143 of the request object.

success

A `boolean` that is `true` if all record changes invoked on the resource by `saveChanges( )` were successful. For more information, see the description of the success property on page 202 of the request object.

request

A reference to the request object returned after the `saveChanges( )` method completed execution and returned all results from its record changes on the server. For more information, see the description of the request object on page 37.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance.

Example

The following code fragment subscribes the function, `onAfterSaveChanges`, to handle the `afterSaveChanges` event fired on the JSDO, `myjsdo`, for changes saved on an OpenEdge single-table resource, `ttCustomer` without using a Submit operation:

```
/* subscribe to event */
myjsdo.subscribe( 'afterSaveChanges', onAfterSaveChanges );
/* some code that initiates multiple CRUD operations and
   sends them to the server */
var newrec = myjsdo.add();
. . .

var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.remove();};

myjsdo.saveChanges(); /* invoked without Submit */

function onAfterSaveChanges( jsdo , success , request ) {
    /* number of resource operations invoked by saveChanges() */
    var lenErrors,
        errors,
        errorType;
    if (success) {
        /* all resource operations invoked by saveChanges() succeeded */
        /* for example, redisplay records in list */
        jsdo.foreach( function(jsrecord) {
            /* reference the record/field as jsrecord.data.<fieldName> */
        });
    }
    else {
        /* handle all operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "Server Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
                case default:
                    errorType = null; // Unexpected errorType value
                    break;
            }
            if (errorType) { /* log all error text
                console.log("ERROR: " + errorType + errors[idxError].error);
                if (errors[idxError].id) { /* error with record object */
                    console.log("RECORD ID: " + errors[idxError].id);
                    /* possibly log the data values for the record with this ID
                       from the matching request.batch.operations[index].jsrecord */
                }
                if (errors[idxError].responseText) {
                    console.log("HTTP FULL TEXT: "
                                + errors[idxError].responseText);
                }
            }
            else { /* unexpected errorType */
                console.log("UNEXPECTED ERROR TYPE: "
                            + errors[idxError].type);
            }
        }
    }
};
```

If successful, the example loops to display the data from every record in JSDO memory (not shown). If unsuccessful, the example loops through the array of error objects returned from the call to `saveChanges( )` (the result of calling `jsdo.ttCustomer.getErrors( )`) and logs annotated values for each error object returned, depending on its type (`errors[idxError].type`). If a record ID is included for a given error object, you can display the field values for the associated record (not shown) that you can return from `request.batch.operations[`*`index`*`].jsrecord`, where *`index`* identifies a returned request object for an operation in the `request.batch.operations` array and the error record ID matches `getId(request.batch.operations[`*`index`*`].jsrecord)`.

Note: This example takes advantage of the default setting of the JSDO `autoApplyChanges` property (`true`), which automatically accepts or rejects changes to JSDO memory based on whether all CUD operations were successful. Note also that for an unsuccessful result, the error information returned by `getErrors( )` is still available until the next invocation of `fill( )` or `saveChanges( )`.

See also:

beforeSaveChanges event on page 90, saveChanges( ) method on page 174, subscribe( ) method (JSDO class) on page 200, unsubscribe( ) method (JSDO class) on page 206

afterUpdate event

Fires after the JSDO, by means of a `saveChanges( )` call following an `assign( )` call, sends a request to update a record in the Data Object resource and receives a response to this request from the server.

This event fires after the response from an Update operation request (sent without using Submit) has been received, and fires after the response from a Submit operation request for each one of possibly multiple record updates has been received.

Note: A single Update operation request is sent for each record object that you have updated in the JSDO with a single call to `saveChanges( )` or `saveChanges(false)`. A single Submit operation request for any and all created, updated, and deleted JSDO record objects is sent with a single call to `saveChanges(true)`.

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , record , success , request )
```

`jsdo`

A reference to the JSDO that invoked the update request. For more information, see the description of jsdo property on page 143 of the request object.

`record`

A reference to the table record upon which the update request acted. For more information, see the description of jsrecord property on page 143 of the request object.

success

A boolean that is true if the update request was successful. For more information, see the description of success property on page 202 of the request object.

request

A reference to the request object returned after the update request completes. For more information, see the description of request object on page 37.

Application code can subscribe a callback to this event by invoking the subscribe( ) method on a JSDO instance or one of its table references.

Example

The following code fragment subscribes the function, `onAfterUpdate`, to handle the `afterUpdate` event fired on the JSDO, `myjsdo`, created for an OpenEdge single-table resource, `ttCustomer`, where `myid` is the known ID of a record to find and update:

```
/* subscribe to event */
myjsdo.subscribe( 'afterUpdate', onAfterUpdate );

/* some code that might update one or more
   records, then apply the updates on the server */
var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.assign( updatedDataObject );};
. . .

myjsdo.saveChanges();

function onAfterUpdate ( jsdo , record , success , request ) {
    var myField,
        lenErrors,
        errors,
        errorType;
    if (success) {
        /* for example, get the values updated by the server from the record
           to redisplay */
        var newValue = record.data.myField;
        . . .
    }
    else { /* Handle Update operation errors */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "BI Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
            }
            /* Log error type and message for current error */
            console.log("ERROR: " + errorType + errors[idxError].error);
            if (errors[idxError].id) { /* Log info for error with record object */
                console.log("RECORD ID: " + errors[idxError].id);
                /* If the error record ID matches the operation
                   record ID, log the record data values */
                if  (errors[idxError].id == record.getId()) {
                    console.log ("Customer Record Fields:");
                    console log ("    CustNum = " + record.data.CustNum);
                    . . .
                }
            }
            /* Log additional HTTP text for web or other server errors */
            if (errors[idxError].responseText) {
                console.log("HTTP FULL TEXT: " + errors[idxError].responseText);
            }
        }
    } /* End Update operation errors */
};
```

See also:

assign( ) method (JSDO class) on page 79, beforeUpdate event on page 91, saveChanges( ) method on page 174, subscribe( ) method (JSDO class) on page 200, unsubscribe( ) method (JSDO class) on page 206

assign( ) method (JSDO class)

Updates field values for the specified `JSRecord` object in JSDO memory.

The specified record object can be either the working record of a JSDO table reference or any record provided by a `JSRecord` object.

To synchronize the change on the server, call the `saveChanges( )` method.

Alias: `update( )`

Return type: `boolean`

Applies to: progress.data.JSRecord class on page 35, progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Syntax

```
jsrecord-ref.assign ( update-object )
jsdo-ref.assign ( update-object )
jsdo-ref.table-ref.assign ( update-object )
```

jsrecord-ref

A reference to a `JSRecord` object for a table record in JSDO memory.

You can obtain a `JSRecord` object by:

- Invoking a JSDO method that returns record objects from a JSDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a JSDO table reference that already has a working record.
- Accessing the *`record`* parameter passed to the callback of a JSDO `afterCreate`, `afterDelete`, or `afterDelete` event.
- Accessing each record object provided by the `jsrecords` property on the *`request`* object parameter passed to the callback of a JSDO `afterSaveChanges` event, or passed to the callback of any Promise object returned from the `saveChanges( )` method. The `jsrecords` property is only available on completion of a Submit operation (`saveChanges(true)`) on a resource that supports before-imaging, and **not** if the resource supports before-imaging without Submit.

jsdo-ref

A reference to the JSDO. You can call the method on *`jsdo-ref`* if the JSDO has only a single table reference, and that table reference has a working record.

table-ref

A table reference on the JSDO that has a working record.

update-object

Passes in the data to update the specified record object in JSDO memory. Each property of the object has the name of a table field and the value to set for that field in the specified record. Any table fields without corresponding properties in *update-object* remain unchanged in the record.

Note: After this method updates the specified record object, and if you have set up automatic sorting using the `autoSort` property, all the record objects for the affected table reference are sorted accordingly. If the sorting is done using sort fields, any `string` fields are compared according to the value of the `caseSensitive` property.

Example

The following code fragment shows a jQuery event defined on a save button to save the current field values for a customer detail form to the corresponding `eCustomer` record in JSDO memory:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );

$('#btnSave').bind('click', function(event) {
  var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
  if (jsrecord) {jsrecord.assign(update-object);};
  dataSet.saveChanges();
});
```

The form has been displayed with previous values of the same record. When the button is clicked, the event handler uses the `findById( )` method to find the original record with the matching internal record ID (`jsrecord`) and invokes the `assign( )` method on `jsrecord` with an object parameter to update the fields in `eCustomer` with any new values entered into the form.

See also:

autoSort property on page 82, caseSensitive property on page 93, getSchema( ) method on page 120, saveChanges( ) method on page 174

async property

A `boolean` that indicates, if set to `true`, that the Data Object resource operation was executed asynchronously in the mobile app.

Data type: `boolean`

Access: Read-only

Applies to: request object on page 37

The `async` property is available only for the following JSDO events or in the request object returned to a jQuery Promise callback:

- `afterCreate`
- `afterDelete`
- `afterFill`
- `afterInvoke`

- `afterUpdate`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe or register to JSDO events, any returned jQuery Promises, and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO. This object is also returned as the value of any JSDO invocation method that you execute synchronously.

See also:

add( ) method on page 49, remove( ) method on page 171, fill( ) method on page 97, invocation method on page 134, invoke( ) method on page 136, saveChanges( ) method on page 174

authenticationModel property (JSDOSession class)

Returns a `string` constant that was passed as an option to the object's class constructor, and specifies the authentication model that the current `JSDOSession` object requires to start a JSDO login session in the web application for which the `JSDOSession` object was also created.

Data type: `string`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 25

Values that can be returned include:

- **`progress.data.Session.AUTH_TYPE_ANON`** — The web application supports Anonymous access. No authentication is required. This is the default value if none is passed to the `JSDOSession` constructor.
- **`progress.data.Session.AUTH_TYPE_BASIC`** — The web application supports HTTP Basic authentication and requires a valid username and password. To have the `JSDOSession` object manage access to the web application's resources for you, you need to pass these credentials in a call to the `JSDOSession` object's `login( )` method. Typically, you would require the user to enter their credentials into a login dialog provided by your mobile app, either using a form of your own design or using a template provided by Progress Software.
- **`progress.data.Session.AUTH_TYPE_FORM`** — The web application uses HTTP Form-based authentication. Like HTTP Basic, Form-based authentication requires user credentials for access to protected resources; the difference is that the web application itself sends a form to the client to get the credentials. However, when you have the `JSDOSession` object manage access to the web application's resources, you handle Form-based authentication the same way that you handle Basic—get the user's credentials yourself and pass them to the `login( )` method. The `JSDOSession` intercepts the form sent by the web application and handles the authentication without that form being displayed.

If the web application requires authentication, you must set this value correctly in the `JSDOSession` constructor to ensure that users can log in.

See also:

The constructor description for the progress.data.JSDOSession class on page 25

autoApplyChanges property

A `boolean` on a JSDO that indicates if the JSDO automatically accepts or rejects changes to JSDO memory when you call the `saveChanges( )` method.

When set to `true`, and after you have invoked the `saveChanges( )` method, the JSDO automatically accepts all changes to JSDO memory that are successfully applied for a given resource operation on the server, and rejects all changes in JSDO memory that are associated with the same resource operation that are completed with an error.

The default setting is `true`.

You can set this property both during JSDO instantiation and on an existing JSDO.

Data type: `boolean`

Access: Readable/Writable

Applies to: progress.data.JSDO class on page 12

When set to `false`, you must invoke one of the following methods at the appropriate time to accept or reject the changes in JSDO memory:

- `acceptChanges( )`
- `acceptRowChanges( )`
- `rejectChanges( )`
- `rejectRowChanges( )`

You typically invoke one of these methods either in the appropriate event handler for a JSDO event or in the appropriate Promise method associated with execution of the `saveChanges( )` method. For information on detecting successful and unsuccessful resource operations, where you might invoke these methods, see the description of the `saveChanges( )` method.

Example

The following code fragment sets the property both when the JSDO is instantiated and after it is instantiated:

```
var jsdoCustomers = new progress.data.JSDO( { autoApplyChanges : false } );
. . .
jsdoCustomers.autoApplyChanges = true;
```

See also:

acceptChanges( ) method on page 44, acceptRowChanges( ) method on page 47, rejectChanges( ) method on page 165, rejectRowChanges( ) method on page 168, saveChanges( ) method on page 174

autoSort property

A `boolean` on a JSDO and its table references that indicates if record objects are sorted automatically on the affected table references in JSDO memory at the completion of a supported JSDO operation.

When set to `true`, and after you have specified a sorting method for each affected table reference, record objects are sorted after the JSDO operation completes its update of JSDO memory. When set to `false`, or if no sorting method is specified for a given table reference, no automatic sorting occurs after the JSDO operation completes. The default setting is `true` for all table references of a JSDO.

Data type: `boolean`

Access: Readable/Writable

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

When set on a JSDO, the property setting affects the sorting of record objects for all table references in the JSDO. When set on a single table reference, the property setting affects the sorting of record objects only for the specified table reference. For example, to set this property to `true` on only a single table reference in the JSDO:

1. Set the value on the JSDO to `false`, which sets `false` on all its table references.
2. Set the value on the selected table reference to `true`, which sets `true` on only the this one table reference.

In order to activate automatic sorting for an affected table reference, you must invoke one of the following JSDO methods to specify a sorting method for the table reference:

- `setSortFields( )` — Identifies the *sort fields* to use in the record objects and whether each field is sorted in ascending or descending order according to its data type. Any `string` fields specified for a table reference are sorted using letter case according to the setting of the `caseSensitive` property (`false` by default).

Note: Changing the value of the `caseSensitive` property triggers an automatic sort if the `autoSort` property is also set to `true` for the affected table reference.

- `setSortFn( )` — Identifies a *sort function* that compares two record objects according to the criteria you specify and returns a value that indicates if one record sorts later than the other in the sort order, or if the two records sort at the same position in the sort order. The `caseSensitive` property setting has no effect on the operation of the specified sort function unless you choose to involve the setting of this property in your criteria for comparison.

If you specify both sort fields and a sort function to sort the record objects for a table reference, the sort function takes precedence. You can also call the `setSortFields( )` and `setSortFn( )` functions to clear one or both settings of the sort fields and sort function. However, at least one setting must be active for automatic sorting to occur on a table reference.

The following supported JSDO operations trigger automatic sorting on any affected table references before they complete their updates to JSDO memory:

- Invoking the `add( )` method — Sorts the record objects of the affected table reference.
- Invoking the `addRecords( )` method — Sorts the record objects of either the single affected table reference or all affected table references in the JSDO. (Unaffected table references do not participate in the sort, including those for which `autoSort` is `false`, those for which no sort fields or sort function are set, or those other than the single JSDO table reference on which `addRecords( )` is called, if it is called only on a single table reference.)
- Invoking the `addLocalRecords( )` method — Sorts the record objects of all affected table references in the JSDO. (Unaffected table references do not participate in the sort, including those for which `autoSort` is `false` and those for which no sort fields or sort function are set.)
- Invoking the `assign( )` method— Sorts the record objects of the affected table reference.
- Assigning a value to a field reference directly on the working record of a table reference (*`jsdo-ref.table-ref.field-ref = value`*) — Sorts the record objects of the affected table reference.

Note: Assignment to a field referenced on the `data` property never triggers automatic sorting (for example, *`jsdo-ref.table-ref`*`.data.`*`field-ref = value`*)

- Changing the value of the `caseSensitive` property — Sorts the record objects of the affected table reference, or of all affected table references if the property value is changed on the JSDO.
- Invoking either the `acceptChanges( )` or `rejectChanges( )` method — Sorts the record objects of all affected table references in the JSDO. (Unaffected table references do not participate in the sort, including any table references for which `autoSort` is `false`, or for which no sort fields or sort function are set.)
- Invoking either the `acceptRowChanges( )` or `rejectRowChanges( )` method — Sorts the record objects of the affected table reference.
- Invoking the `fill( )` method — Sorts the record objects of all affected table references in the JSDO. (Unaffected table references do not participate in the sort, including any table references for which `autoSort` is `false`, or for which no sort fields or sort function are set.)

Note: Invoking the `remove( )` method does not trigger an automatic sort and has no effect on any existing sort order established for the table reference. However, if there is a sort order that depends on the presence or absence of the record object you are removing, and you want to establish the appropriate sort order when this record object is absent, you must manually sort the remaining record objects using the `sort( )` method by passing it the same sort function that you used to establish the sort order when this record object was present.

Caution: Because automatic sorting executes in JavaScript on the client side, sorting a large set of record objects can take a significant amount of time and make the UI appear to be locked. You might set a wait or progress indicator just prior to any action that can sort a large record set to alert the user that the app is working.

Example

In the following code fragment, automatic local sorting is turned off for all table references of the `dsCustOrds` JSDO by setting its `autoSort` property to `false`. Automatic sorting is then turned on for the `eCustomer` table reference of the JSDO by setting its `autoSort` value to `true` and using the `setSortFields( )` method to set its `Name` field as the single, descending sort field:

```
dsCustOrds = new progress.data.JSDO( { name: 'dsCustomerOrders' });
dsCustOrds.autoSort = false.
dsCustOrds.eCustomer.autoSort = true.
dsCustOrds.eCustomer.setSortFields( "Name:DESC" );
dsCustOrds.fill();
. . .
```

When the `fill( )` method executes on the JSDO, all the referenced tables are loaded from the Progress Application Server into JSDO memory with their record objects already sorted in case-insensitive, primary key order (by default). The record objects for `eCustomer` are then sorted locally in case-insensitive, descending order of the `Name` field.

See also:

acceptChanges( ) method on page 44, acceptRowChanges( ) method on page 47, add( ) method on page 49, addRecords( ) method on page 61, assign( ) method (JSDO class) on page 79, caseSensitive property on page 93, fill( ) method on page 97, rejectChanges( ) method on page 165, rejectRowChanges( ) method on page 168, setSortFields( ) method on page 194, setSortFn( ) method on page 195, sort( ) method on page 197, table reference property (JSDO class) on page 203

batch property

A reference to an object with a property named `operations`, which is an array containing the request objects for each of the one or more record-change operations on a resource performed in response to calling the JSDO `saveChanges( )` method without using Submit (either with an empty parameter list or with the single parameter value of `false`).

Data type: `Object`

Access: Read-only

Applies to: request object on page 37

In addition to the properties documented for the request object, each request object returned in the `operations` property array also contains an `operation` property with a `number` value that indicates the record-change operation associated with the request object:

- **1** — Create
- **3** — Update
- **4** — Delete

The `batch` property is available only for the following JSDO events or in the request object returned to a Promise object callback, and **only** after calling `saveChanges( )` either with an empty parameter list or with the single parameter value of `false` to invoke individual Data Object Create, Update, or Delete (CUD) operations:

- `afterSaveChanges`
- `beforeSaveChanges`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe both to JSDO events and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO.

See also:

saveChanges( ) method on page 174, jsrecords property on page 144

beforeCreate event

Fires before the JSDO, by means of a `saveChanges( )` call following an `add( )` call, sends a request to create a record in the Data Object resource on the server.

This event fires before each Create operation request is sent without using Submit, and fires before each one of possibly multiple record creates are sent with a single Submit operation request.

Note: A single Create operation request is sent for each record object that you have created in the JSDO with a single call to `saveChanges( )` or `saveChanges(false)`. A single Submit operation request for any and all created, updated, and deleted JSDO record objects is sent with a single call to `saveChanges(true)`.

Applies to:progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , record , request )
```

jsdo

A reference to the JSDO that is invoking the create request. For more information, see the description of jsdo property on page 143 of the request object.

record

A reference to the table record upon which the create request is about to act. For more information, see the description of jsrecord property on page 143 of the request object.

request

A reference to the request object before the create request is sent. For more information, see the description of request object on page 37.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance or one of its table references.

Example

The following code fragment subscribes the function, `onBeforeCreate`, to handle the `beforeCreate` event fired on the JSDO, `myjsdo`, created for a single-table resource, by assigning data to the newly created record before sending it to the server:

```
/* subscribe to event */
myjsdo.subscribe( 'beforeCreate', onBeforeCreate );

/* some code that might add one or more
   records and save them on the server */
var jsrecord = myjsdo.add();

. . .

myjsdo.saveChanges();

function onBeforeCreate( jsdo , record , request ) {
    /* for example, here you might update data in the record
       before it is sent to the server to be created */
    record.assign( { myField1 = myvalue, myField2 = myvalue2 } );
};
```

See also:

add( ) method on page 49, afterCreate event on page 64, record property on page 165, saveChanges( ) method on page 174, subscribe( ) method (JSDO class) on page 200, unsubscribe( ) method (JSDO class) on page 206

beforeDelete event

Fires before the JSDO, by means of a `saveChanges( )` call following a `remove( )` call, sends a request to delete a record in the Data Object resource on the server.

This event fires before each Delete operation request is sent without using Submit, and fires before each one of possibly multiple record deletes are sent with a single Submit operation request.

Note: A single Delete operation request is sent for each record object that you have deleted in the JSDO with a single call to `saveChanges( )` or `saveChanges(false)`. A single Submit operation request for any and all created, updated, and deleted JSDO record objects is sent with a single call to `saveChanges(true)`.

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , record , request )
```

jsdo

A reference to the JSDO that is invoking the delete request. For more information, see the description of jsdo property on page 143 of the request object.

record

A reference to the table record upon which the delete request is about to act. For more information, see the description of jsrecord property on page 143 of the request object.

request

A reference to the request object before the delete request is sent. For more information, see the description of request object on page 37.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance or one of its table references.

Example

The following code fragment subscribes the function, `onBeforeDelete`, to handle the `beforeDelete` event fired on the JSDO, `myjsdo`, created for a single-table resource, where `myid` is the known ID of a record to find and delete:

```
/* subscribe to event */
myjsdo.subscribe( 'beforeDelete', onBeforeDelete );

/* some code that might delete one or more
   records and remove them on the server */
var jsrecord = myjsdo.findById( myid );
if (jsrecord) {jsrecord.remove();};

. . .

myjsdo.saveChanges();

function onBeforeDelete( jsdo , record , request ) {
    /* code to execute before sending delete request to the server */

};
```

See also:

afterDelete event on page 67, remove( ) method on page 171, saveChanges( ) method on page 174, subscribe( ) method (JSDO class) on page 200, unsubscribe( ) method (JSDO class) on page 206

beforeFill event

Fires on the JSDO before a call to the `fill( )` method executes and sends a Read operation request to its Data Object resource.

Alias: `beforeRead`

Applies to: progress.data.JSDO class on page 12

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , request )
```

jsdo

A reference to the JSDO that is invoking the Read operation on the resource. For more information, see the description of the jsdo property on page 143 of the request object.

request

A reference to the request object before the Read operation request is sent to the resource. For more information, see the description of the request object on page 37.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance.

Example

The following code fragment subscribes the callback function, `onBeforeFill`, to handle the `beforeFill` event fired on the JSDO, `myjsdo`:

```
/* subscribe to event */
myjsdo.subscribe( 'beforeFill', onBeforeFill );

myjsdo.fill();

function onBeforeFill ( jsdo , request ) {
    /* for example, do any preparation to receive data from the server */
};
```

See also:

afterFill event on page 69, fill( ) method on page 97, subscribe( ) method (JSDO class) on page 200, unsubscribe( ) method (JSDO class) on page 206

beforeInvoke event

Fires when a custom invocation method is called asynchronously on a JSDO before the request for the Invoke operation is sent to the server.

Synchronous invocation method calls do not cause this event to fire.

Applies to: progress.data.JSDO class on page 12

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , request )
```

jsdo

A reference to the JSDO that is invoking the method. For more information, see the description of jsdo property on page 143 of the request object.

request

A reference to the request object returned before the operation begins. For more information, see the description of request object on page 37.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance.

Note: To subscribe a handler for this event, the `subscribe( )` method requires that you pass, as a parameter, the name of the invocation method to which the event applies.

Example

The following code fragment subscribes the function, `onBeforeInvokeGetOrderTotalAndStatus`, to handle the `beforeInvoke` event fired on the JSDO, `dataSet`, for an invocation of the `getOrderTotalAndStatus( )` invocation method passed the parameters specified by `paramObject`:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );
                  dataSet.subscribe( 'beforeInvoke', 'getOrderTotalAndStatus',
                  onAfterInvokeGetOrderTotalAndStatus );

                  dataSet.getOrderTotalAndStatus( paramObject );

                  function onAfterInvokeGetOrderTotalAndStatus ( jsdo , request ) {
                      /* code to execute before sending request to the server */
                  };
```

See also:

afterInvoke event on page 70, invocation method on page 134, subscribe( ) method (JSDO class) on page 200, unsubscribe( ) method (JSDO class) on page 206

beforeSaveChanges event

Fires once on the JSDO before a call to the `saveChanges( )` method sends the first request to create, update, or delete a record in its Data Object resource on the server.

This event fires before the first of possibly multiple Create, Update, and Delete operation requests are sent without using Submit, and fires before the first one of possibly multiple record changes are sent with a single Submit operation request.

Note: A single Create, Update, or Delete operation request is sent for each record object that you have created, updated, or deleted (respectively) in the JSDO with a single call to `saveChanges( )` or `saveChanges(false)`. A single Submit operation request for any and all created, updated, and deleted JSDO record objects is sent with a single call to `saveChanges(true)`.

Applies to: progress.data.JSDO class on page 12

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , request )
```

jsdo

A reference to the JSDO that is about to invoke the `saveChanges( )` method for one or more record changes on its resource. For more information, see the description of jsdo property on page 143 of the request object.

request

A reference to the request object before the first record-change request is sent to the resource. For more information, see the description of request object on page 37.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance.

Example

The following code fragment subscribes the function, `onBeforeSaveChanges`, to handle the `beforeSaveChanges` event fired on the JSDO, `myjsdo`, where `myid` is the known ID of a record to find and process for resource operations being sent to the server:

```
/* subscribe to event */
myjsdo.subscribe( 'beforeSaveChanges', onBeforeSaveChanges );

/* some code that initiates multiple CUD requests and
   sends them to the server */
var newrec = myjsdo.add();
var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.remove();};

. . .

myjsdo.saveChanges();

function onBeforeSaveChanges ( jsdo , request ) {
    /* code to execute before sending first (or only) request to the server */
};
```

See also:

afterSaveChanges event on page 72, saveChanges( ) method on page 174, subscribe( ) method (JSDO class) on page 200, unsubscribe( ) method (JSDO class) on page 206

beforeUpdate event

Fires before the JSDO, by means of a `saveChanges( )` call following an `assign( )` call, sends a request to update a record in the Data Object resource on the server.

This event fires before each Update operation request is sent without using Submit, and fires before each one of possibly multiple record updates are sent with a single Submit operation request.

Note: A single Update operation request is sent for each record object that you have updated in the JSDO with a single call to `saveChanges( )` or `saveChanges(false)`. A single Submit operation request for any and all created, updated, and deleted JSDO record objects is sent with a single call to `saveChanges(true)`.

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdo , record , request )
```

jsdo

A reference to the JSDO that is invoking the update request. For more information, see the description of jsdo property on page 143 of the request object.

record

A reference to the table record upon which the update request is about to act. For more information, see the description of jsrecord property on page 143 of the request object.

request

A reference to the request object before the update request is sent. For more information, see the description of request object on page 37.

Application code can subscribe a callback to this event by invoking the `subscribe( )` method on a JSDO instance or one of its table references.

Example

The following code fragment subscribes the function, `onBeforeUpdate`, to handle the `beforeUpdate` event fired on the JSDO, `myjsdo`, created for a single-table resource, where `myid` is the known ID of a record to find and update. In this case, the `onBeforeUpdate` event callback assigns additional data to the updated record before sending it to the server:

```
/* subscribe to event */
myjsdo.subscribe( 'beforeUpdate', onBeforeUpdate );

/* some code that might update one or more
   records and save them on the server */
var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.assign( { myField1 = myvalue, myField2 = myvalue2 } );};

. . .

myjsdo.saveChanges();

function onBeforeUpdate( jsdo , record , request ) {
    /* for example, here you might update data in the record
       further before it is sent to the server */
    record.assign( { myField4 = myvalue4, myField5 = myvalue5 } );
};
```

See also:

assign( ) method (JSDO class) on page 79, afterUpdate event on page 76, saveChanges( ) method on page 174, subscribe( ) method (JSDO class) on page 200, unsubscribe( ) method (JSDO class) on page 206

caseSensitive property

A `boolean` on a JSDO and its table references that indicates if `string` field comparisons performed by supported JSDO operations are case sensitive or case-insensitive for the affected table references in JSDO memory.

When set to `true`, all supported comparisons on `string` fields for an affected table reference are case sensitive. When set to `false`, all supported comparisons on `string` fields for an affected table reference are case insensitive. The default setting is `false` for all table references of a JSDO.

Note: This default setting (case insensitive) matches the default setting for letter case comparison in OpenEdge Data Objects (ABL).

Data type: `boolean`

Access: Readable/Writable

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

When set on a JSDO, the property setting affects all table references in the JSDO. When set on a single table reference, the property setting affects only the specified table reference. For example, to set this property to `true` on only a single table reference in the JSDO:

1. Set the value on the JSDO to `false`, which sets `false` on all its table references.
2. Set the value on the selected table reference to `true`, which sets `true` on only the one table reference.

The JSDO operations that follow this property setting in `string` field comparisons include:

- Sorting record objects in JSDO memory, including automatic sorting using sort fields that you specify using the `autoSort` property and the `setSortFields( )` method, and manual sorting using specified sort fields that you perform using the `sort( )` method

  ---

  Note: Changing the value of this property triggers an automatic sort if the `autoSort` property is also set to `true` for the affected table reference.

- Merging record objects into JSDO memory for all merge modes that perform record field comparisons during execution of the `addRecords( )` method

---

Note: Any default `string` field comparisons that you might do in JavaScript within the callback functions that you specify for other JSDO methods and events are always case sensitive according to JavaScript rules and ignore this property setting.

Note: To conform to Unicode default letter case mapping, the JSDO support for case-insensitive `string`-field comparison and sorting relies on the `toUpperCase( )` JavaScript function instead of the `toLocaleUpperCase( )` JavaScript function. The latter function uses the locale letter case mapping, which might be different from the default letter case mapping in Unicode.

Note: Unlike character string comparisons in OpenEdge ABL, all JSDO-supported `string` field comparisons **include** trailing spaces and **ignore** any OpenEdge-specified collation tables.

Example

In the following code fragment, automatic local sorting is set up for the `eCustomer` table reference on a JSDO created for an OpenEdge resource (`dsCustOrds`), with its `Name` field as the single descending sort field. All other table references on `dsCustOrds` have no automatic local sorting set up by default. Because OpenEdge sorting on `string` fields is case-insensitive by default, the fragment makes the local sort on the `Name` field case sensitive by setting `caseSensitive` on `eCustomer` to `true`:

```
dsCustOrds = new progress.data.JSDO( { name: 'dsCustomerOrders' } );

dsCustOrds.autoSort = false;
dsCustOrds.eCustomer.autoSort = true;
dsCustOrds.eCustomer.setSortFields( "Name:descending" );
dsCustOrds.eCustomer.caseSensitive = true;

dsCustOrds.fill();
. . .
```

When the `fill( )` method executes on the JSDO, after all the referenced tables are loaded from the server, with their record objects already sorted in case-insensitive, primary key order (by default), the record objects for `eCustomer` are then sorted locally in case-sensitive, descending order by the `Name` field.

See also:

addRecords( ) method on page 61, autoSort property on page 82, setSortFields( ) method on page 194, sort( ) method on page 197

catalogURIs property

Returns the list of URIs successfully used to load Data Service Catalogs into the current `JSDOSession` object.

Data type: `string` array

Access: Read-only

Applies to: progress.data.JSDOSession class on page 25

This list includes the URI for each Data Service Catalog loaded using the `addCatalog( )` method on the current `JSDOSession` or `Session` object. To return a corresponding list of Data Object Services for which the Data Service Catalogs are loaded, read the `services` property.

See also:

addCatalog( ) method (JSDOSession class) on page 50, services property on page 190

clientContextId property

The value of the most recent client context identifier (CCID) that the current `JSDOSession` or `Session` object has found in the `X-CLIENT-CONTEXT-ID` HTTP header of a server response message.

If none has yet been found, the value is `null`.

Data type: `string`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 25

The JSDOSession object automatically detects, stores, and returns the CCID sent by any appropriately configured web application for which it has started a JSDO login session.

Note: You can set up an OpenEdge web application to send a CCID when configuring the web application to use SSO. For more information, depending on the OpenEdge application server you use, see the sections on enabling SSO for a web application in the administration documentation for the Progress Application Server for OpenEdge.

Note: For an OpenEdge resource, this CCID is the same as the value of the ClientContextId property on the ABL Progress.Lang.OERequestInfo object that is passed from an application server client (in this case, the web application hosting the Data Object Service) to the server process that is executing a Data Object Service request. You can access this OERequestInfo object from the resource Business Entity using the ABL CURRENT-REQUEST-INFO attribute on the ABL SESSION system handle. This CCID value is also available as the ABL SESSION-ID attribute of the single sign-on (SSO) client-principal handle returned by the GetClientPrincipal( ) method of the same ABL OERequestInfo class-based object.

---

connected property

Returns a boolean that indicates the most recent online status of the current JSDOSession object when it last determined if the web application it manages was available.

If the property value is true, the object most recently determined that the session is connected and logged in to its web application. If its value is false, the session was last found to be disconnected. The default value is false.

Note: Because of the dynamics of any network environment, the value of this property might not reflect the current status of the object's connection to its web application. You can therefore invoke the object's ping( ) method (either explicitly or automatically by setting the value of its pingInterval property) to update the object's most recent online status.

Data type: boolean

Access: Read-only

Applies to: progress.data.JSDOSession class on page 25

The most recent session online status determination might be identified from any of the following:

- A successful result of the JSDOSession object executing its login( ) method, which sets the property to true. Prior to calling login( ) for the first time, the value of this property is false.
- A successful result of the JSDOSession object executing its logout( ) or invalidate( ) method, which sets the property to false.
- The JSDOSession object receiving an offline or online event from its window object.
- A JSDO attempting to send a request to an OpenEdge Data Object Service that the JSDOSession object manages.
- The result of the JSDOSession object executing its ping( ) method.

See also:

login( ) method (JSDOSession class) on page 145, offline event on page 155, online event on page 157, ping( ) method (JSDOSession class) on page 160

data property

The data (field) and state values for a record associated with a `JSRecord` object.

Data type: `Object`

Access: Read-only

Applies to: progress.data.JSRecord class on page 35

The returned object contains a field reference property (*`field-ref`* in syntax) for each field (column) in the table, where the property name is identical to a table field name and the property value for the corresponding JavaScript data type.

You can obtain a `JSRecord` object by invoking one of the JSDO methods that returns record objects from a JSDO table reference (`find( )`, `findById( )`, or `foreach( )`) or by accessing the `record` property on a JSDO table reference that already has a working record.

Note: If a given JSDO table has a working record, you can access each *`field-ref`* of the working record directly on the corresponding table reference property (*`table-ref`*) of the JSDO. For the working record of a table reference, then, references to the `JSRecord` object of the working record and its `data` property are both implied by the table reference alone.

Caution: Never write directly to a to a *`field-ref`* using this `data` property; in this case, use *`field-ref`* only to read the data. Writing field values using the `data` property does not mark the record for update when calling the `saveChanges( )` method, nor does it re-sort the record in JSDO memory according to any order you have established using the `autoSort` property. To mark a record for update and automatically re-sort the record according to the `autoSort` property, you must assign a field value either by setting a *`jsdo-ref.table-ref.field-ref`* for a working record or by calling the `assign( )` method on a valid *`table-ref`* or `JSRecord` object reference. For information on table references (*`table-ref`*), see the reference entry for the table reference property (JSDO).

See also:

assign( ) method (JSDO class) on page 79, autoSort property on page 82, find( ) method on page 104, findById( ) method on page 106, foreach( ) method on page 108, record property on page 165, table reference property (JSDO class) on page 203

deleteLocal( ) method

Clears out all data and changes stored in a specified local storage area, and removes the cleared storage area.

Return type: `undefined`

Applies to: progress.data.JSDO class on page 12

Syntax

```
deleteLocal ( [ storage-name ] )
```

storage-name

The name of the local storage area to be removed. If *storage-name* is not specified, blank, or `null`, the name of the default storage area is used. The name of this default area is `jsdo_`*serviceName*`_`*resourceName*, where *serviceName* is the name of the Data Object Service that supports the JSDO instance, and *resourceName* is the name of the resource (table, dataset, etc.) for which the JSDO instance is created.

If this method encounters any errors, it leaves the specified storage area unchanged and throws an exception.

Example

The following code fragment clears out all the data currently stored in the default storage area and removes the storage area:

```
dataSet = new progress.data.JSDO( 'dsStaticData' );
dataSet.fill();
dataSet.saveLocal();
.
.
.
dataSet.deleteLocal();
```

See also:

acceptChanges( ) method on page 44, rejectChanges( ) method on page 165, saveChanges( ) method on page 174, saveLocal( ) method on page 188

fill( ) method

Initializes JSDO memory with record objects from the data records in a single-table resource, or in one or more tables of a multi-table resource, as returned by the Read operation of the Data Object resource for which the JSDO is created.

This method also causes an `offline` or `online` event to fire if it detects that there has been a change in the JSDO login session's online state.

This method always executes asynchronously and returns results (either or both) in subscribed JSDO event callbacks or in callbacks that you register using methods of a Promise object returned as the method value. A Promise object is always returned as the method value if jQuery Promises (or the exact equivalent) are supported in your development environment. Otherwise, this method returns `undefined`.

Alias: `read( )`

Return type: jQuery Promise or `undefined`

Applies to: progress.data.JSDO class on page 12

Working record: After completing execution, the working record for each JSDO table is set to its first record, depending on any active parent-child relationships (for a multi-table resource) and automatic sort settings. So, for each child table, the first record object is determined by its table reference sort order (if any) and its relationship to the related working record in its parent table.

Syntax

```
fill ( [ parameter-object | filter-string ] )
```

parameter-object

An `Object` initialized with a set of properties that can be used on the server depending on a JSDO MappingType plugin that is both registered on the client and for which a corresponding Read operation is defined in the server Data Object. For more information on JSDO Mapping Type plugins and how to register and use them, see the description of the progress.data.PluginManager class on page 36.

The JSDO also provides a built-in MappingType plugin named `"JFP"` that can be used if the server Data Object is built to support it. This `"JFP"` plugin allows you to specify properties that can be used on the server to select, sort, and page the records to be returned. This built-in plugin supports any JSDO accessed by the JSDO dialect of the Kendo UI DataSource or any JSDO involved in data access. If you are using the JSDO dialect of the Kendo UI DataSource to bind a JSDO instance to Kendo UI widgets, the DataSource initializes and invokes `fill( )` with specific *`parameter-object`* properties that are defined based on the settings of the DataSource `serverPaging`, `serverFiltering`, and `serverSorting` configuration properties. Also, if OpenEdge Data Objects are being accessed as external objects, as well as in calls to standard objects from Kendo UI, `fill( )` is invoked with similar *`parameter-object`* properties based on requirements.

Note: Upon execution, the `fill( )` method immediately passes the *`parameter-object`* to a `requestMapping( )` function in the plugin to perform conversions.

The `fill( )` method further converts these *`parameter-object`* property settings to a format that is customized for each server Data Object resource, depending on its type. This format is then applied to the resource Read operation based on the presence of a `mappingType` property set to `"JFP"` in the Data Service Catalog. For information on the requirements and the effects of using a `"JFP"` MappingType plugin for a Read operation in OpenEdge resources, see the topics on updating Business Entities for access by Kendo UI DataSources and external objects in *OpenEdge Development: Web Services*. For internal objects accessed using the JSDO, the requirements for using the built-in `"JFP"` MappingType plugin are managed internally by the server.

The *`parameter-object`* properties required for the built-in `"JFP"` MappingType plugin, which you can initialize when you call `fill( )` directly, include:

- **`filter`** — An `Object` containing property settings used to select the records to be returned. These property settings are in the same format as the property settings in the Kendo UI DataSource `filter` property object. For more information, see the `filter` configuration property description in the Kendo UI DataSource documentation. The format that the `fill( )` method uses to pass this selection criteria to the server is specified, again, in the Data Service Catalog using the `mappingType` property.

  If a `filter` object is not specified, the resource Read operation determines the records to return without selection criteria sent from the client.

- **`id`** — A `string` specifying a unique ID that the resource understands to identify a specific record.

Note: This property is not currently used by the Kendo UI.

- **skip** — A number that specifies how many records to skip before returning (up to) a page of data. You must specify this property together with the top property.
- **sort** — An expression that specifies how to sort the records to be returned using one of the following formats:
  - An Object with property settings in the same format as the property settings in the Kendo UI DataSource sort property object. For more information, see the sort configuration property description in the Kendo UI DataSource documentation.
  - An Array of strings in the same format as the *sort-fields* parameter of the setSortFields( ) method.

  If this property is not specified, the resource Read operation determines the order of records to return without sort information sent from the client.
- **tableRef** — A string specifying the name of a table reference in the JSDO. This property is required when the JSDO represents a multi-table resource and the filter property Object is **also** specified with filter information.
- **top** — A number that specifies how many records (the page size) to return in a single page of data after using skip. You must specify this property together with the skip property.

  Note: The final page of a larger result set can contain a smaller number of records than top specifies.

Note: You must specify both skip and top to implement server paging. For example, if you want to return the 5th page of data with a page size (top) of 10, set skip to 40. If these properties are not specified together, the resource Read operation determines the records to return without using any paging information sent from the client.

Note: If you access the JSDO using only the JSDO dialect of the Kendo UI DataSource, the DataSource invokes the fill( ) method with appropriate settings for these properties based on the corresponding Kendo UI settings.

filter-string

This is a string containing selection criteria used by the Read operation as required by the resource:

- **For an OpenEdge resource** — This is a string that the resource defines for use on an OpenEdge application server to select records to be returned, much like the WHERE option of the ABL record phrase. The actual format of this string and its affect on the records returned is determined by the ABL routine that implements the resource Read operation. For example, you might pass:
  - A single key value (e.g., "30")
  - A relational expression (e.g., "CustNum > 10 AND CustNum < 30")
  - An actual WHERE string (e.g., 'Item.CatDescription CONTAINS "ski & (gog* ! pol*)"')

Note: For an OpenEdge resource, the JSDO requires the URI for the Read operation of the resource to contain the following query string: "?*filter*=~{*filter*~}", where *filter* is the name of a string input parameter defined for the ABL routine that implements the operation (INPUT *filter* AS CHARACTER).

Caution: Using an actual WHERE string for a dynamic ABL query can create a potential security issue.

Note: If you want to use sorting or paging that your server Data Object resource supports, you must pass *parameter-object* instead of *filter-string*.

If you do not pass either *parameter-object* or *filter-string* to this method, the records returned depend entirely on the Read operation that the resource implements.

Promise method callback signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that fill( ) returns:

Syntax:

```
promise.done( function ( jsdo , success , request ) )
promise.fail( function ( jsdo , success , request ) )
promise.always( function ( jsdo , success , request ) )
```

promise

A reference to the Promise object that is returned as the value of the fill( ) method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDO class on page 12.

jsdo

A reference to the JSDO that invoked the fill( ) method (Read operation) on its resource. For more information, see the description of the jsdo property on page 143 of the request object.

success

A boolean that is true if the Read operation was successful. For more information, see the description of the success property on page 202 of the request object.

Note: It is not always necessary to test the value of *success* in a Promise method callback for the fill( ) method, especially if the callback is registered using *promise*.done( ) and *promise*.fail( ), where the callback executes according to this value.

request

A reference to the request object returned after the fill( ) method completed execution and returned any results from its Read operation on the server. For more information, see the description of the request object on page 37.

General operation

This method invokes the Read operation on the resource defined for the current JSDO. The result of calling this method replaces any prior data in JSDO memory with the record objects returned by this Read operation. The record objects are stored in one or more JSDO tables that correspond to the table or tables defined for the resource. If the JSDO is accessing multi-table resource (with one or more tables), such as an OpenEdge ProDataSet, and the resource supports before-imaging, the JSDO also updates the state of its JSDO memory with any before-image data sent with the loaded record objects, including any changes to record objects recorded in their before-image data.

Caution: If the JSDO has pending record changes from the client that you want to save on the server, do not call this method before you call the JSDO `saveChanges( )` method. Otherwise, the pending changes will be lost when JSDO memory is initialized with records from the Read operation.

Note: After this method initializes JSDO memory with record objects, and if you have set up automatic sorting using the `autoSort` property, the record objects of each affected JSDO table reference are sorted in JSDO memory according to the sort order you have established for the JSDO. If sorting is done using sort fields, any `string` fields are compared according to the value of the `caseSensitive` property on a given table reference. If the `autoSort` property setting is `false` for a given table reference, its record objects are loaded in the order that they were serialized from the corresponding resource table.

Returning and handling results

This method returns results asynchronously in two different ways, and in the following order, depending on the development environment used to build the mobile app:

1. **Using JSDO named events for all environments** — The following events fire before and after the `fill( )` method executes, respectively, with results handled by callback functions that you subscribe as documented for each event:
   a. beforeFill event on page 88
   b. afterFill event on page 69
2. **Using a Promise object returned for environments that support jQuery Promises** — Any callbacks that you register using Promise object methods all execute both **after** the `fill( )` method itself and **after** any subscribed `afterFill` event callbacks complete execution. Note that the signatures of all Promise method callbacks match the signature of the `afterFill` event callback function so you can specify an existing `afterFill` event callback as the callback function that you register using any Promise method.

Because the callbacks that you register with any returned Promise methods execute only after all subscribed `afterFill` event callbacks complete execution, you can invoke logic in the Promise method callbacks to modify any processing done by the event callbacks.

If the `fill( )` method completes successfully, the *`success`* parameter for any `afterFill` event callback or Promise method and the `success` property of each handler's *`request`* parameter object are both set to `true`, and any data records returned by the resource Read operation are loaded into JSDO memory. Otherwise, both the *`success`* parameter and `success` property are set to `false`, and you can read any error results by calling the `getErrors( )` method on a single JSDO table reference, or by inspecting the setting of the `response` property in the same *`request`* parameter object.

Note: When returning before-image data from an OpenEdge ProDataSet resource, it is possible (though rare) for record-change errors that result from server update activity to be returned as well. The `fill( )` method does not return an unsuccessful result in this case. If you need to identify before-image errors in records returned by a Read operation, you can query the record objects in JSDO memory and call `getErrorString( )` on each record object to identify any such errors that were returned.

You can read any record objects loaded into JSDO memory by `fill( )` using the `find( )`, `findById( )`, `foreach( )`, and `getData( )` methods of the JSDO. You can return the schema for this data by using the `getSchema( )` method. You can create a new record object in JSDO memory using the JSDO `add( )` method, and you can update or delete a single record object in JSDO memory by using the `assign( )` or `remove( )` method, respectively. You can display a record in a UI form that you dynamically bind to the record by calling the `display( )` method on a class instance. You can merge data returned by an invocation method with the data in JSDO memory using the `addRecords( )` method, and you can store and merge records in JSDO memory both to and from a local storage location using the `saveLocal( )`, `readLocal( )`, and `addLocalRecords( )` methods.

Example

The following code fragment shows the `fill( )` method invoked on a JSDO for an OpenEdge single-table resource (`dsCustomerOrder`), with results returned using the `afterFill` event:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );
dataSet.subscribe("afterFill", onAfterFill);

dataSet.fill();

function onAfterFill( jsdo , success , request ) {
    var lenErrors,
        errors,
        errorType;

    if (success) {
        /* for example, add code to display all records on a list */
        jsdo.foreach(function (jsrecord) {
            /* you can reference the fields as jsrecord.data.field */
        });
    }
    else {
        /* handle Read operation errors */
        errors = jsdo.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
            }
            console.log("READ ERROR: " + errorType + errors[idxError].error);
            if (errors[idxError].responseText) {
                console.log("HTTP FULL TEXT: " + errors[idxError].responseText);
            }
        }
    }
};
```

Note that for an OpenEdge resource, the `getErrors( )` method can return one or more error messages for a Read operation in an array of error objects.

The following code fragment shows the `fill( )` method invoked on a JSDO for a similar OpenEdge single-table resource (`dsCustomerOrder`), with results returned using a Promise object:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );

dataSet.fill().done(
    function( jsdo, success, request ) {
        /* for example, add code to display all records in a list */
        jsdo.foreach(function (jsrecord) {
            /* you can reference the fields as jsrecord.data.<fieldName> */
        });
    }).fail(
    function( jsdo, success, request ) {
        var lenErrors,
        errors,
        errorType;

        /* handle Read operation errors */
        errors = jsdo.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
            }
            console.log("READ ERROR: " + errorType + errors[idxError].error);
            if (errors[idxError].responseText) {
                console.log("HTTP FULL TEXT: " + errors[idxError].responseText);
            }
        }
    }
});
```

Using a Promise object, the Promise `done` and `fail` functions do not have to test the `success` parameter for a successful (`true`) or failed (`false`) execution of the `fill( )` method, because `done` executes only when `fill( )` succeeds and `fail` executes only when `fill( )` fails.

The following code fragment shows the `fill( )` method invoked on a JSDO for a resource (`Orders`), with results returned using a Promise object:

```
dataSet = new progress.data.JSDO( 'Orders' );

dataSet.fill().done(
    function( jsdo, success, request ) {
        /* for example, add code to display all records on a list */
        jsdo.foreach(function (jsrecord) {
            /* you can reference the fields as jsrecord.data.<fieldName> */
        });
    }).fail(
    function( jsdo, success, request ) {
        var lenErrors,
        errors,
        errorType;

        /* handle operation errors */
        errors = jsdo.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            console.log("READ ERROR: " + errors[idxError].error);
            if (errors[idxError].responseText) {
                console.log("HTTP FULL TEXT: " + errors[idxError].responseText);
            }
        }
    });
```

Any error results from a Read operation are returned by `getErrors( )` as the single error type `progress.data.JSDO.ERROR` with the `errors[idxError].error` property set to a single returned `string` value.

See also:

autoSort property on page 82, caseSensitive property on page 93, getErrors( ) method on page 110, getId( ) method on page 116, invocation method on page 134, progress.data.PluginManager class on page 36, response property on page 172, saveChanges( ) method on page 174, setSortFields( ) method on page 194, success property on page 202

find( ) method

Searches for a record in a table referenced in JSDO memory and returns a reference to that record if found. If no record is found, it returns `null`.

Return type: progress.data.JSRecord class on page 35

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Working record: After completing execution, any record found becomes the working record for the associated table. If the searched table has child tables in the same multi-table resource, and the `useRelationships` property is `true`, the working record of the result set for each child is set to the first record as determined by the relationship to its respective parent. If a record is not found, the working record is not set, and the working records of any child tables are also not set.

Syntax

```
jsdo-ref.find ( funcRef )
jsdo-ref.table-ref.find ( funcRef )
```

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference.

table-ref

A table reference on the JSDO.

funcRef

A reference to a JavaScript callback function that returns a `boolean` value and has the following signature:

Syntax:

```
function [ func-name ] ( jsrecord-ref )
```

Where *func-name* is the name of a callback function that you define external to the `find( )` parameter list and *jsrecord-ref* is a `JSRecord` reference to the next available record on the specified table reference. You can then pass *func-name* to the `find( )` method as the *funcRef* parameter. Alternatively, you can specify *funcRef* as the entire inline function definition without *func-name*.

The `find( )` method executes your *funcRef* callback for each record of the table reference, until it returns `true`, indicating that the callback has found the record. You can then test the field values on the `data` property of *jsrecord-ref* to determine the result. Otherwise, your callback returns `false` and the `find( )` method executes the callback again on the next available record.

If your *funcRef* callback finds the record, `find( )` completes execution with both its return value and the `record` property of the associated table reference set to the `JSRecord` reference of the found working record. If `find( )` reaches the end of the available records without *funcRef* returning `true`, `find( )` completes execution with both its return value and the `record` property on the table reference set to `null`, indicating that the sought for record was not found.

If the associated table reference is for a child table in a multi-table resource, if the `useRelationships` property is `true`, `find( )` uses the relationship to filter out all but the child records of the working record in the parent table. However, if the working record of the parent is not set, `find( )` throws an error. If `useRelationships` is `false`, the search includes all records of the child table and no error is thrown.

Example

In following code fragment, `jsdo` references a single `customer` table:

```
var jsdo = new progress.data.JSDO( 'customer' );

jsdo.find(function(jsrecord) {
  return (jsrecord.data.CustNum == 10);
});
```

The inline function passed to `find( )` returns `true` or `false` based on the value of the `CustNum` property of the object returned by the `data` property for the currently available `JSRecord` reference.

See also:

data property on page 96, foreach( ) method on page 108, record property on page 165

findById( ) method

Locates and returns the record in JSDO memory with the internal ID you specify.

If no record is found, it returns `null`. You can access the internal ID of the working record of a table reference, or any `JSRecord` object, by calling the `getId( )` method on the object returned by the `data` property of the `JSRecord`.

Return type: progress.data.JSRecord class on page 35

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Working record: After completing execution, any record found becomes the working record for the associated table. If the searched table has child tables in the same multi-table resource, and the `useRelationships` property is `true`, the working record of the result set for each child is set to the first record as determined by the relationship to its respective parent. If a record is not found, the working record is not set, and the working records of any child tables are also not set.

Syntax

```
jsdo-ref.findById ( id )
jsdo-ref.table-ref.findById ( id )
```

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference.

table-ref

A table reference on the JSDO.

id

The internal record ID used to match a record of the table reference. This is the same value originally returned for the record using the `getId( )` function. It is typically used to create a jQuery listview

row to display the record or a detail form used to display the record in the current HTML document. Later, when a listview row or detail form is selected, the corresponding `id` attribute with this value can be used to return the record from the JSDO, possibly to update the record with new data values input by the user.

If `findById( )` locates a record with the matching record ID, it completes execution with both its return value and the `record` property of the table reference set to the `JSRecord` reference of the found working record. If the function does not locate the record, it completes execution with both its return value and the `record` property on the table reference set to `null`, indicating that no record of the table reference has a matching internal record ID.

If the table reference references a child table in a multi-table resource, when the `useRelationships` property is `true`, `findById( )` uses the relationship to filter out all but the child records of the working record in the parent table; the remaining child records are excluded from the search. If `useRelationships` is `false` or the working record of the parent is not set, the search includes all records of the child table and no error is thrown.

Example

The following code fragment shows a jQuery event defined on a save button to save the current field values for a customer detail form to the corresponding `eCustomer` record in JSDO memory:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );

$('#btnSave').bind('click', function(event) {
  var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
  if (jsrecord) {jsrecord.assign();};
  dataSet.saveChanges();
});
```

The form has been displayed with previous values of the same record. When the button is clicked, the event handler finds the original `eCustomer` record by calling `findById( )` with the `id` attribute of the form (`$('#custdetail #id').val()`), which is set to the internal ID of the record. The `jsrecord.assign( )` method then updates the record from the values of the corresponding form fields and `saveChanges( )` invokes the resource `"update"` operation on the application server to save the updated record to its data source.

See also:

data property on page 96, foreach( ) method on page 108, getId( ) method on page 116

fnName property

For an Invoke operation, the name of the custom JSDO invocation method that executed the operation.

The `fnName` property is `null` in the case of a request object returned by a resource Create, Read, Update, Delete, or Submit operation.

Data type: `string`

Access: Read-only

Applies to: request object on page 37

The `fnName` property is available only for the following JSDO event or in the request object returned to a jQuery Promise callback:

- `afterInvoke`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe or register to JSDO events, the jQuery Promise returned from an `invoke( )` method, and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO. This object is also returned as the value of any JSDO invocation method that you execute synchronously.

Note: The value of the `fnName` property is the same as that of the *`op-name`* parameter passed to the `subscribe( )` method that subscribed to the current invoke operation event.

See also:

invocation method on page 134, invoke( ) method on page 136, subscribe( ) method (JSDO class) on page 200

foreach( ) method

Loops through the records of a table referenced in JSDO memory and invokes a user-defined callback function as a parameter on each iteration.

With each iteration, it also sets the current record as the working record and passes it as a parameter to the callback function. This function can then operate on the working record and return a value indicating whether the `foreach( )` terminates the loop or invokes the callback function on the next working record of the table.

If the referenced table has child tables in the same multi-table resource, and the `useRelationships` property is `true`, with each iteration through the loop, the working record of the result set for each child is set to the first record as determined by the relationship to its respective parent.

Return type: `null`

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Working record: After completing execution, the working records of the associated table, and any child tables, are the most recent working records established when the method terminates the loop.

Syntax

```
jsdo-ref.foreach ( funcRef )
jsdo-ref.table-ref.foreach ( funcRef )
```

`jsdo-ref`

A reference to the JSDO. You can call the method on *`jsdo-ref`* if the JSDO has only a single table reference.

`table-ref`

A table reference on the JSDO.

funcRef

A reference to a JavaScript callback function that returns a `boolean` value and has the following signature:

Syntax:

```
function [ func-name ] ( jsrecord-ref )
```

Where *func-name* is the name of a callback function that you define external to the `foreach( )` parameter list and *jsrecord-ref* is a `JSRecord` object reference to the next working record on the table reference. You can then pass *func-name* to the `foreach( )` method as the *funcRef* parameter. Alternatively, you can specify *funcRef* as the entire inline function definition without *func-name*.

The `foreach( )` method executes your *funcRef* callback for each record of the table reference, making this record the working record and passing it in as *jsrecord-ref*. You can then access the field values of the working record using the `data` property on *jsrecord-ref* or any field references available from the table reference. You can also invoke other JSDO methods, for example, to operate on the working record, including additional calls to `foreach( )` to operate on working records of any child tables.

Your *funcRef* callback can terminate the `foreach( )` loop by returning `false`. If the callback does not return `false`, the loop continues.

If the table reference references a child table in a multi-table resource, when the `useRelationships` property is `true`, `foreach( )` uses the relationship to filter out all but the child records of the working record in the parent table. However, if the working record of the parent is not set, `foreach( )` throws an error. If `useRelationships` is `false`, the loop includes all records of the child table and no error is thrown.

Example

After creating a JSDO for a `dsCustomer` resource and loading it with record objects, the following code fragment shows the `foreach( )` method looping through `eCustomer` records in JSDO memory and displaying the `CustNum` and `Name` fields from each record, one record per line, to the current HTML page, and also to the console log:

```
jsdo = new progress.data.JSDO({ name: 'dsCustomer' });
jsdo.subscribe( 'AfterFill', onAfterFillCustomers, this );

jsdo.fill();

function onAfterFillCustomers(jsdo, success, request) {
  jsdo.eCustomer.foreach( function(customer) {
    document.write(customer.data.CustNum + ' ' + customer.data.Name + '<br>');
    console.log(customer.data.CustNum + ' ' + customer.data.Name);
  } );
};
```

See also:

data property on page 96, find( ) method on page 104, progress.data.JSRecord class on page 35

getData( ) method

Returns an array of record objects for a table referenced in JSDO memory.

If this is a child table in a multi-table resource, and the `useRelationships` property is `true`, the specific record objects in the result set depends on the relationship to its parent.

Return type: `Object` array

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Syntax

```
jsdo-ref.getData ( )
jsdo-ref.table-ref.getData ( )
```

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference.

table-ref

A table reference on the JSDO.

See also:

getSchema( ) method on page 120

getErrors( ) method

Returns an array of errors from the most recent invocation of Create, Read, Update, Delete, or Submit operations (CRUD or Submit) that you have invoked by calling the JSDO `fill( )` or `saveChanges( )` method on a Data Object resource.

You can call this method on a single JSDO table reference at any point after `fill( )` or `saveChanges( )` is invoked, typically in a callback function executed for each fired JSDO `after*` event or for any Promise object returned by `fill( )` or `saveChanges( )`. The errors returned from this method apply either to the data in the single JSDO table that is involved in the invoked operation or operations, or to the web or application servers that handle the requests. These errors are cleared with the next invocation of `fill( )` or `saveChanges( )` on the same resource, no matter the setting of the JSDO `autoApplyChanges` property.

This method allows you to access error information returned for every possible error, whether it is generated by:

- Routines on the application server that implement the CRUD or Submit operations on a given Data Object resource, and whether or not the resource supports before-imaging
- The application server that implements the specified Data Object Service, regardless of the Data Object resource and its executed CRUD or Submit operations
- The specified web server or web application, regardless of the Data Object Service you are accessing

Note: Using this method, you have no need to inspect error information returned by either the `response` property or the `xhr` property (XMLHttpRequest object) in any request object returned by `fill( )` or `saveChanges( )`, or the error string returned by the `getErrorString( )` method called on a `JSRecord` object.

Note: If you have a reference to the JSDO used to instantiate the JSDO dialect of the Kendo UI DataSource, you can also call this method in any callback function registered for the `error` event on the DataSource.

Return type: Array of `Object`

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Syntax

```
jsdo-ref.getErrors ( )
jsdo-ref.table-ref.getErrors ( )
```

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference.

table-ref

A table reference on the JSDO.

Response

The array returned by this method contains errors associated with the resource operation or operations that were invoked by the most recent call to `fill( )` or `saveChanges( )`. Each element of the array returned by this method is an `Object` that corresponds to a single type of error returned by an operation on the resource and contains the following properties:

- **`error`** — A `string` value that describes the error depending on the error type. For more information, see the `type` property of this `Object`.
- **`errorNum`** — A `numeric` value that returns an error number. This property is returned only if the error type is `progress.data.JSDO.APP_ERROR`. For more information, see the `type` property of this `Object`.
- **`id`** — A `string` value containing the internal ID of the record object on which the resource operation has returned the error. This property can be returned, if appropriate, for any error type **except** `progress.data.JSDO.ERROR`. If the error type is `progress.data.JSDO.APP_ERROR` and is returned for a resource Read operation, the `id` property is **never** returned. If the error type is `progress.data.JSDO.DATA_ERROR` and is returned for a resource CUD or Submit operation, the `id` property is **always** returned. How you can use this value to return information about the record object depends on the error type. For more information, see the `type` property of this `Object`.
- **`responseText`** — A `string` value that contains the complete text of the HTTP response when the error type is `progress.data.JSDO.ERROR`. For example, this can be the HTML text of an error page displayed by the browser, which you can also display using the browser's web inspector. For more information, see the `type` property of this `Object`.
- **`type`** — A numeric constant that identifies the origin of the error returned by this `Object`:
  - **`progress.data.JSDO.DATA_ERROR`** — An error returned for a CUD or Submit operation on a record object in a resource with before-image data. The response for this type of error uses HTTP status code

200 OK, and the error information is returned using `prods` properties of the ProDataSet object in the response.

For this error type, the `error` property returns an error string associated with the record object and this record object is identified by the `id` property. For a Submit operation, you can look up the corresponding before-image data for the record by invoking the `getId( )` method on each element of the array returned by the `jsrecords` property on the returned request object. You can also find the current data for the record in JSDO memory, if it exists, using the JSDO `findById( )` method.

Note: From an OpenEdge Data Object resource that supports before-imaging, this error type can be returned by setting both the ABL `ERROR` and `ERROR-STRING` attributes on a buffer object handle set to the corresponding record in the corresponding ABL temp-table. For more information on these attributes, see *OpenEdge Development: ABL Reference* and *OpenEdge Development: ProDataSets*.

- **`progress.data.JSDO.APP_ERROR`** — An HTTP 4xx or 5xx error returned for a CRUD operation on a resource **without** before-image data (and sometimes also for a CUD or Submit operation **with** before-image data).

  For this error type, the `error` property returns an error message associated with the CRUD operation and the `errorNum` property returns the numeric value of the error number. For a CUD operation, the associated record object in the resource can also be identified using the value of the `id` property. You can also find the current data for the record in JSDO memory, if it exists, using the JSDO `findById( )` method.

  Note: From an OpenEdge Data Object resource, this error type can be returned using a constructor or method of the built-in ABL class, `Progress.Lang.AppError`. For more information, see *OpenEdge Development: ABL Reference*.

  Other properties are available to interpret this information, depending on the resource operation and event callback where `getErrors( )` is called, and include:

  - In the callback for a CUD operation event (`afterCreate`, `afterUpdate`, or `afterDelete`), you can also return the record object identified by the `id` property as the value of the `jsrecord` property on the request object returned by the callback.
  - In the callback for an `afterSaveChanges` event on completion of a call to `saveChanges(false)` (**without** Submit), you can identify the request object returned for each CUD operation by inspecting the value of the `batch` property on the request object returned by the callback and compare this information with the information and data returned by the `error` property and `id` property for this error type.
  - In the callback for an `afterSaveChanges` event on completion of a call to `saveChanges(true)` (**with** Submit), if this error type is returned, it is typically not returned with an `id` property, as the error type does not apply to a particular record object.
  - In the callback for an `afterFill` event (on completion of a Read operation invoked by a call to `fill( )`), the `id` property is never returned, as the error type does not apply to a particular record object.

  Note: The same options that apply to callbacks for `afterSaveChanges` and `afterFill` events also apply to corresponding callbacks registered to returned Promise objects.

- **`progress.data.JSDO.RETVAL`** — An HTTP 4xx or 5xx error that returns a CRUD operation return value on a resource **without** before-image data (and sometimes also for a CUD or Submit operation **with** before-image data).

For this error type, the `error` property returns the CRUD operation return value as a string. For a CUD operation, the associated record object in the resource can also be identified by the `id` property. You can find the current data for the record in JSDO memory, if it exists, using the JSDO `findById( )` method.

Note: From an OpenEdge Data Object resource, this error type can be returned using the `RETURN ERROR` statement, or using a constructor or method of the built-in ABL class, `Progress.Lang.AppError`. For more information, see *OpenEdge Development: ABL Reference*.

Note: OpenEdge Data Object Services using the WebHandler service provider can return multiple `RETVAL` errors per operation in the array returned by `getErrors( )`, including one for the operation itself and one for each routine on the operation call stack. Any `RETVAL` error, regardless of the Data Object Service, can be followed by one or more `APP_ERROR` errors.

- **`progress.data.JSDO.ERROR`** — The text of a JavaScript exception or a web server HTTP 4xx or 5xx error returned without regard to any particular CRUD or Submit operation that was invoked.

  For an HTTP 4xx or 5xx error, the `error` property returns an HTTP error string in the following form:

  ```
  HTTP error-number error-text
  ```

  Where:

  - *`error-number`* is the HTTP 4xx or 5xx numeric value. For example, 500.
  - *`error-text`* is a brief description of the web server error. For more information, you can inspect the value of the `responseText` property or view the associated web page displayed by the browser web inspector.

If there are no errors for the operation and resource on which `getErrors( )` is called, this method returns an empty array (`[]`).

Note: For a CUD or Submit operation on a resource with before-image data that can return a `DATA_ERROR`, `APP_ERROR`, or `RETVAL` error, the operation can return **either** a `DATA_ERROR` error **or** it can return a `RETVAL` error followed by one or more `APP_ERROR` errors, but not **both**.

Note: If the method returns any errors, the JSDO also returns the request object for the operation with its `success` property set to `false`. This is also the value of the *`success`* parameter passed to any callback function subscribed to a JSDO `after*` event or passed to any callback function registered by a JSDO Promise object returned by `fill( )` or `saveChanges( )`.

Note: This method does not track errors returned by an Invoke operation.

Example

The following code fragment shows the `getErrors( )` method invoked on the JSDO in the callback function (`onAfterSaveChanges`) that is subscribed to the JSDO `afterSaveChanges` event after invoking a Submit operation (`saveChanges(true)`) on the JSDO OpenEdge ProDataSet resource named `'Customer'` with a `ttCustomer` table, and which supports both before-imaging and the Submit operation:

```
dsCustomer = new progress.data.JSDO({ name: 'Customer' });
dsCustomer.subscribe('afterFill', onAfterFillCustomers, this);
dsCustomer.subscribe('afterSaveChanges', onAfterSaveChanges, this);
. . .

dsCustomer.saveChanges(true); /* Invoke Submit */
. . .

function onAfterSaveChanges(jsdo, success, request) {
    var errors;
    var errorType;
    console.log("DEBUG: AfterSaveChanges: " + success + " errors: " + errors.length);

    if (!success) {
        errors = jsdo.ttCustomer.getErrors();
        for (var i = 0; i < errors.length; i++) {
            switch(errors[i].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "Server Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #" + errors[i].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
                case default:
                    errorType = null; // Unexpected errorType value
                    break;
            }
            if (errorType) {
                console.log("ERROR: " + errorType + errors[i].error);
                if (errors[i].id) { /* error with record object */
                    console.log("RECORD ID: " + errors[i].id);
                    /* Possibly log record change information based on data error
                       record object found in the request.jsrecords using getId()
                    */
                }
                if (errors[i].responseText) {
                    console.log("HTTP FULL TEXT: "
                                + errors[i].responseText);
                }
            }
            else { /* unexpected errorType */
                console.log("UNEXPECTED ERROR TYPE: "
                            + errors[i].type);
            }
        }
    }
}
```

Note: If the `'Customer'` resource supports before-imaging but does not support a Submit operation, this same code fragment continues to work by changing the `saveChanges( )` method call from `saveChanges(true)` to `saveChanges(false)` (or simply, `saveChanges()`). However, in this case, there is no `request.jsrecords` property available for access. Instead, you might locate information on the record and its record change operation by inspecting the `request.batch` property.

See also:

autoApplyChanges property on page 82, batch property on page 85, getErrorString( ) method on page 115, fill( ) method on page 97, findById( ) method on page 106, getId( ) method on page 116, jsrecord property on page 143, progress.data.JSRecord class on page 35, response property on page 172, saveChanges( ) method on page 174, xhr property on page 209

getErrorString( ) method

Returns any before-image error string in the data of a record object referenced in JSDO memory that was set as the result of a resource Create, Update, Delete, or Submit operation.

If there is no error string in the data of the specified record object, this method returns `undefined`.

Note: This error string can be returned from an OpenEdge Data Object resource that supports before-imaging by setting both the ABL `ERROR` and `ERROR-STRING` attributes on a buffer object handle set to the corresponding record in the corresponding ABL temp-table. For more information on these attributes, see *OpenEdge Development: ABL Reference* and *OpenEdge Development: ProDataSets*.

The specified record object can be either the working record of a referenced table, or any record provided by a `JSRecord` object.

Return type: `string`

Applies to: progress.data.JSRecord class on page 35, progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Syntax

```
jsrecord-ref.getErrorString ( )
jsdo-ref.getErrorString ( )
jsdo-ref.table-ref.getErrorString ( )
```

jsrecord-ref

A reference to a `JSRecord` object for a table record in JSDO memory.

You can obtain a `JSRecord` object by:

- Invoking a JSDO method that returns record objects from a JSDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a JSDO table reference that already has a working record.
- Accessing the *`record`* parameter passed to the callback of a JSDO `afterCreate`, `afterDelete`, or `afterDelete` event.

- Accessing each record object provided by the `jsrecords` property on the *`request`* object parameter passed to the callback of a JSDO `afterSaveChanges` event, or passed to the callback of any Promise object returned from the `saveChanges( )` method. The `jsrecords` property is only available on completion of a Submit operation (`saveChanges(true)`) on a resource that supports before-imaging, and **not** if the resource supports before-imaging without Submit.

`jsdo-ref`

A reference to the JSDO. You can call the method on *`jsdo-ref`* if the JSDO has only a single table reference, and that table reference has a working record.

`table-ref`

A table reference on the JSDO that has a working record.

The error string returned by this function contains error information only for a Data Object Service operation on a resource that supports before-imaging. This information can only be returned for a resource Create, Update, Delete, or Submit operation (`saveChanges(true)`) that affects the before-image data of a resource record. The JSDO stores the error information from each affected resource record in the corresponding JSDO record object as an internal property of its data with a reserved name.

Note: This method is most useful when the `autoApplyChanges` property is `false`. When `autoApplyChanges` is `true`, the method automatically accepts or rejects record changes, and clears all associated error conditions and information after the final `after*` event is fired and handled (if handled) for a given operation.

See also:

addRecords( ) method on page 61, afterCreate event on page 64, afterDelete event on page 67, afterUpdate event on page 76, autoApplyChanges property on page 82, find( ) method on page 104, findById( ) method on page 106, foreach( ) method on page 108, saveChanges( ) method on page 174, record property on page 165

getId( ) method

Returns the unique internal ID for the record object referenced in JSDO memory.

The specified record object can be either the working record for a referenced table, or any record provided by a `JSRecord` object.

Return type: `string`

Applies to: progress.data.JSRecord class on page 35, progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Syntax

```
jsrecord-ref.getId ( )
jsdo-ref.getId ( )
jsdo-ref.table-ref.getId ( )
```

`jsrecord-ref`

A reference to a `JSRecord` object for a table record in JSDO memory.

You can obtain a JSRecord object by:

- Invoking a JSDO method that returns record objects from a JSDO table reference (find( ), findById( ), or foreach( ))
- Accessing the record property on a JSDO table reference that already has a working record.
- Accessing the *record* parameter passed to the callback of a JSDO afterCreate, afterDelete, or afterDelete event.
- Accessing each record object provided by the jsrecords property on the *request* object parameter passed to the callback of a JSDO afterSaveChanges event, or passed to the callback of any Promise object returned from the saveChanges( ) method. The jsrecords property is only available on completion of a Submit operation (saveChanges(true)) on a resource that supports before-imaging, and **not** if the resource supports before-imaging without Submit.

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference, and that table reference has a working record.

table-ref

A table reference on the JSDO that has a working record.

The internal record ID returned by this function is a unique value generated by the JSDO for each record object loaded into JSDO memory using the fill( ), add( ), or addRecords( ) methods. The JSDO adds this value to the data of each record object in a property with a reserved name. The JSDO uses this record ID to map the record objects in JSDO memory to the HTML elements of mobile apps.

To return and set the specified record as the working record, you can pass any value returned by this method to the findById( ) method called on the associated table reference.

Note: The value assigned to the internal record ID for any given record object can change with each invocation of the fill( ) method.

See also:

add( ) method on page 49, addRecords( ) method on page 61, fill( ) method on page 97, findById( ) method on page 106

getPlugin( ) method

Returns a reference to the registered JSDO plugin object with the specified name.

Return type: Object

Applies to: progress.data.PluginManager class on page 36

Syntax

```
progress.data.PluginManager.getPlugin( name )
```

name

A `string` expression that evaluates to the name of the plugin. If a plugin by this name does not exist, the method returns `undefined`.

Example

The JSDO provides a built-in `"JFP"` plugin with a `requestMapping` function that supports the JSON Filter Pattern (JFP). The following example dynamically modifies this built-in plugin to handle the response from a particular OpenEdge Data Object resource, as follows:

```
jsdo = new progress.data.jsdo( . . . );

var defaultPlugin = progress.data.PluginManager.getPlugin("JFP");
defaultPlugin.responseMapping =

    function ( jsdo, response, info ) {


        var newData = response.dsCustomer.eCustomer;

        for (var i = 0; i < newData.length; i++) {
            newData[i].Country = "USA";
        }

        return response;

    };
```

First, the `getPlugin( )` method returns a reference to this built-in plugin as `defaultPlugin`. A custom `responseMapping` function is then dynamically registered for the plugin that modifies data in the `dsCustomer.eCustomer` table returned from the OpenEdge Data Object resource.

Upon execution, the `fill( )` method performs conversions using the plugin's default `requestMapping` function, then invokes the resource Read operation on the server. When the `response` from the server is returned, this customized `responseMapping` function executes to modify `response` data before it is loaded into JSDO memory.

Note: For the OpenEdge Read operation to make use of this built-in `"JFP"` plugin, the ABL method that implements the operation in the Business Entity (the Data Object source) must be annotated with a `mappingType` property set to `"JFP"`. For more information on annotating this property, see the Notes in the description of the `addPlugin( )` method.

See also

addPlugin( ) method on page 57, fill( ) method on page 97

getProperties( ) method

Returns an object containing the names and values of the user-defined properties defined in the current JSDO instance.

Return type: `Object`

Applies to: progress.data.JSDO class on page 12

Syntax

```
getProperties ( )
```

Example

The following code fragment replaces any existing user-defined properties in `jsdoCustOrders` with the two user-defined properties, `"prop1"` and `"prop2"`. It then calls `getProperties( )` to return an object containing all the JSDO's user-defined properties (`"prop1"` and `"prop2"`) and writes their values to the console log:

```
jsdoCustOrders = new progress.data.JSDO(. . .);

jsdoCustOrders.setProperties({"prop1", 100, "prop2", 500});
var props = jsdoCustOrders.getProperties();
console.log("prop1: " + props.prop1);
console.log("prop2: " + props.prop2);
```

The output from the `console.log` statements in this example appear as follows:

```
prop1: 100
prop2: 500
```

See also:

getProperty( ) method on page 119, setProperties( ) method on page 192

getProperty( ) method

Returns the value of the specified JSDO user-defined property.

Return type: Any

Applies to: progress.data.JSDO class on page 12

Syntax

```
getProperty ( name )
```

name

The name of a user-defined property to query from the JSDO.

Example

The following code fragment writes the value of the user-defined property, `"prop1"`, defined in `jsdoCustOrders` to the console log:

```
jsdoCustOrders = new progress.data.JSDO(. . .);

console.log("prop1: " + jsdoCustOrders.getProperty("prop1");
```

See also:

getProperties( ) method on page 118, setProperty( ) method on page 193

getSchema( ) method

Returns an array of objects, one for each field that defines the schema of a table referenced in JSDO memory.

The properties of each object define the schema elements of the respective field.

Return type: Array of `Object`

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Syntax

```
jsdo-ref.getSchema ( )
jsdo-ref.table-ref.getSchema ( )
```

`jsdo-ref`

A reference to the JSDO. You can call the method on *`jsdo-ref`* if the JSDO has only a single table reference.

`table-ref`

A table reference on the JSDO.

The properties in each `Object` of the array include those defined in the Data Service Catalog for each field of the specified resource table schema, for example:

- `ablType`
- `default`
- `format`
- `name`
- `type`

For more information on these and other properties that can define the schema for a table field in a Data Service Catalog, see the specification for the Cloud Data Object (CDO) Catalog file in:

https://github.com/CloudDataObject/CDO/blob/master/CloudDataObject_Catalog.pdf

See also:

getData( ) method on page 110, fill( ) method on page 97

getSession( ) stand-alone function

A stand-alone function that creates and returns a `progress.data.JSDOSession` instance with a specified JSDO login session already established and a specified Data Service Catalog already loaded.

This function combines the features of:

- Preparing the `JSDOSession` class constructor and instantiating the instance.
- Calling the `login( )` method on the instance to establish a JSDO login session.
- Calling the `addCatalog( )` method on the instance to load a single Data Service Catalog.

The resulting login session is then ready to support creation of JSDO instances for the one or more resources provided by the specified Data Object Service. In addition, this function provides automatic support for page refresh in web apps, if specified, when you call it.

This method is always executed asynchronously and returns results in callbacks that you register using methods of a Promise object returned as the function value.

Note: In order to invoke this function successfully, jQuery Promises must be supported in your development environment. Otherwise, the function throws an exception.

Note: This function does not support proxy servers (servers that function as a security service).

Return type: jQuery Promise

Applies to: The `progress.data` namespace

Syntax

```
progress.data.getSession ( parameter-object )
```

parameter-object

An object that contains the following configuration properties:

- **`serviceURI`** — (Required) A `string` expression containing the URI of the web application for which to start a JSDO login session. This web application must support one or more Data Object Services in order to create JSDOs for the resources provided by the application. This URI is appended with a string that identifies a resource to access as part of the login process.

  If the mobile app from which you are logging in is a hybrid app that will be installed to run directly in a native device container, or if it is a web app deployed to a different web server from the web application that hosts the JSDO login session, you must specify `serviceURI` as an absolute URI to the Tomcat server domain or host and port, for example, `http://www.progress.com/SportsMobileApp`, or perhaps for testing, `http://localhost:8980/SportsMobileApp`.

If the mobile app from which you are logging in is a web app deployed to the same Apache Tomcat server as the web application that hosts the JSDO login session, you can specify `serviceURI` as a relative URI, for example, `/SportsMobileApp`, which is relative to the deployment end point (Tomcat server domain or host and port).

- **`catalogURI`** — (Required) A `string` expression that specifies the URI of a Data Service Catalog file. This URI can specify a location in a web application running on a web server. It is typically the location of the web application that hosts the Data Object Service where the returned `JSDOSession` object has a JSDO login session.

  If the Catalog is deployed to the same web server (or device) that the app UI is loaded from, `catalogURI` can be specified using a relative path, for example, `/SportsMobileApp/static/OrderEntrySvc.json`, where `/SportsMobileApp` is the relative URI of the web application that hosts the Catalog. Otherwise, it must be specified using an absolute path that includes the Tomcat server domain or host and port, for example, `http://www.progress.com:8980/SportsMobileApp/static/OrderEntrySvc.json`, or perhaps for testing, `http://localhost:8980/SportsMobileApp/static/OrderEntrySvc.json`.

  Note: The default Catalog URI for a Catalog created for an OpenEdge Data Object Service, relative to the Apache Tomcat server domain or host and port where the session is logged in, is the following: */WebApplicationName/*`static`*/ServiceName*`.json`, where *`WebApplicationName`* is the name of the web application and *`ServiceName`* is the name of the Data Object Service for which the Data Service Catalog is created.

- **`authenticationModel`** — (Required only for Basic or Form authentication) A `string` constant that specifies one of the three authentication models that the returned `JSDOSession` object supports, depending on the web application configuration:

  - **`progress.data.Session.AUTH_TYPE_ANON`** — (Default) The web application supports Anonymous access. No authentication is required.
  - **`progress.data.Session.AUTH_TYPE_BASIC`** — The web application supports HTTP Basic authentication and requires a valid username and password. To have the `JSDOSession` object manage access to the web application's resources, you need to pass these credentials using the **`username`** and **`password`** properties. Typically, you would require the user to enter their credentials into a login dialog provided by your mobile app, either using a form of your own design or using a template provided by Progress Software Corp.
  - **`progress.data.Session.AUTH_TYPE_FORM`** — The web application uses Form-based authentication. Like HTTP Basic, Form-based authentication requires user credentials for access to protected resources; the difference is that the web application itself sends a form to the client to get the credentials. However, when you have the `JSDOSession` object manage access to the web application's resources, you handle Form-based authentication the same way that you handle Basic—get the user's credentials using the **`username`** and **`password`** properties. The `JSDOSession` intercepts the form sent by the web application and handles the authentication without that form being displayed.

  If the web application requires authentication, you must set this value correctly to ensure that users can log in.

  For more information on these authentication models and how to configure them for a web application, see the sections on configuring web server authentication models in the administration documentation for the server hosting your Data Object Service.

- **`username`** — (Required only for Basic or Form authentication) A `string` expression containing a user ID for the function to send to the web server for authentication.

Note: The `userName` property of the returned `JSDOSession` object contains this value.

- **`password`** — (Required only for Basic or Form authentication) A `string` expression containing a password for the function to send to the web server to authenticate the specified user.
- **`name`** — (Optional) A `string` with an operative value that you define to enable page refresh support for the returned `JSDOSession` object. The operative value can be any value other than the empty string (`""`), `null`, or `undefined`.

  If this page refresh support is enabled and the function successfully logs the newly instantiated `JSDOSession` into its server web application, the object stores the state of its JSDO login session using this `name` property value as a key. Then, at any time after log in, if the user initiates a browser refresh on the web app page, the `JSDOSession` object automatically identifies and restores its login session using this value. This helps to ensure, after a page refresh, that the web app does not need to prompt the user again for credentials in order to re-establish its connection to the web application for this `JSDOSession`.

  If you do not specify an operative value for `name` (the default), or you specify a non-operative value, such as the empty string (`""`), `null`, or `undefined`, the `JSDOSession` is instantiated without this page refresh support.

  For more information on how a login session is restored for a `JSDOSession` object with page refresh support, see the Notes for the progress.data.JSDOSession class on page 25description.

Note: If you pass the `getSession( )` function the same value for `name` as an existing `JSDOSession`, it will return a `JSDOSession` using that same key. Both of them have the potential to overwrite each other. You must ensure that you pass a unique `name` value for each call to `getSession( )` (or the `JSDOSession` constructor).

Note: Page refresh supports **only** Anonymous and Form-based authentication. You cannot enable page refresh support when you set `authenticationModel` for a `JSDOSession` to `progress.data.Session.AUTH_TYPE_BASIC`; in this case, any setting of the `name` property is ignored. To enable page refresh support, you must set the `authenticationModel` property to either `progress.data.Session.AUTH_TYPE_ANON` or `progress.data.Session.AUTH_TYPE_FORM`.

Note: When calling this `getSession( )` function, you have no need to call the `isAuthorized( )` method on the returned `JSDOSession` object in order to test that an authorized login session is established it to manage page refresh. This function automatically manages page refresh for any successful call by a web app.

Note: Similar to the `login( )` method on the `progress.data.JSDOSession` class, the `getSession( )` function always relies on the default web resource, `/static/home.html`, as the login target. In addition, if you are both using Basic authentication and the web app is running on an iOS device, this function waits up to four (4) seconds to time out before failing to authenticate a login session or load a Data Service Catalog.

Promise method signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the function that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that `getSession( )` returns:

Syntax:

```
promise.done( function ( JSDOsession , result ) )
promise.fail( function ( result , info ) )
promise.always( function ( JSDOsession , result | result , info ) )
```

Note: The `always( )` method is always passed what the `done( )` or the `fail( )` methods are passed.

promise

A reference to the Promise object that is returned as the value of the `getSession( )` function. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDOSession class on page 25.

JSDOsession

A reference to the `JSDOSession` object that was successfully created and returned by `getSession( )` with a valid JSDO login session and a single loaded Data Service Catalog.

result

An `integer` constant indicating the overall result of the call that can have one of the following values:

- **`progress.data.Session.SUCCESS`** — The `JSDOSession` was created, logged in, and a Catalog added successfully. You can use JSDOs to access any Data Object Services supported by the web application to which the `JSDOSession` object has logged in.
- **`progress.data.Session.AUTHENTICATION_FAILURE`** — Login failed because of an authentication error. This might be because the JSDO session login had invalid credentials or the authentication error happened when adding the Catalog.
- **`progress.data.Session.GENERAL_FAILURE`** — `JSDOSession` creation failed because of something other than an authentication error.

  Note: This value can also be returned if invalid user credentials triggered a login timeout using Basic authentication on an iOS device.

Note: It is not always necessary to test the value of *`result`* in a Promise method callback for `getSession( )`, especially if the callback is registered using *`promise`*`.done( )`, where the callback always executes with the same value (`progress.data.Session.SUCCESS`).

info

A JavaScript object that can have the following properties:

- **`errorObject`** — Any error object thrown as a result of sending a login or add Catalog request to the web server.

  Note: If this object is thrown because of a timeout triggered according to the four (4) second maximum wait time to login or load a Catalog, the `message` property of the error object is set to `"login timeout expired"`.

- **xhr** — A reference to the XMLHttpRequest object that was used to make the server request to either log in or add a catalog.

General web server interaction

The general web server interaction with and response to this method depends on the authentication model that the web server uses and how resources are accessed and protected. You configure the authentication model for each web application deployed to the Apache Tomcat and specify both the web application URI and its corresponding authentication model to the `getSession( )` function.

For more information on these authentication models and how to configure them for a web application, see the sections on configuring web server authentication models in the server documentation for your Data Object Service.

Caution: You must be sure that security is configured to complete authentication before the application requests resources in the Data Service Catalog. Although it is possible to configure application security so that only the Data Object resources in the Catalog require authentication, Progress does not recommend this approach. Instead, Progress recommends that you require authentication for application resources in addition to those defined in the Catalog, and require that the authentication occur prior to accessing any resources in the Catalog. Once the user is authenticated, the web server provides access to all other resources, including Catalog resources, according to the user's authorization settings.

Example

The following code fragment calls `getSession( )` in a `try` block, handling successful results with the Promise `done( )` method, which invokes one inline function, and handling unsuccessful results with the Promise `fail( )` method, which invokes another inline function. For a successful call, it saves the returned `JSDOSession` and logs a message to the consule, then creates and initializes a JSDO for further processing. For an unsuccessful call, it only logs a message to the console. If the call to `getSession( )` itself throws an exception, this is handled in a `catch` block by logging the associated message to the console:

```
var promise,
    myjsdosession;

try {
    promise = progress.data.getSession({
        name: "mySession",
        authenticationModel: progress.data.Session.AUTH_TYPE_FORM,
        serviceURI: "https://someHost:8810/myWebApplication",
        catalogURI: "https://someHost:8810/myWebApplication/static/Customer.json",
        username: "restuser",
        password: "password"
    });
    promise.done( function( jsdosession, result ) {
        var jsdo;

        myjsdosession = jsdosession;
        console.log("Success on getSession().");

        // create a JSDO
        jsdo = new progress.data.JSDO({
                    name: 'MyCustomerBE'
        });
    });
    promise.fail( function( result, info ) {
        console.log("Failure from getSession()");
    });
}
catch(ex) {
    console.log("Exception: " + ex.message);
};
```

See also:

addCatalog( ) method (JSDOSession class) on page 50, login( ) method (JSDOSession class) on page 145, progress.data.JSDOSession class on page 25, serviceURI property on page 191, userName property on page 209

hasChanges( ) method

Returns `true` if JSDO memory contains any pending changes (with or without before-image data), and returns `false` if JSDO memory has no pending changes.

Return type: `boolean`

Applies to: progress.data.JSDO class on page 12

Syntax

```
hasChanges ( )
```

This method always returns `true` if any change to JSDO memory has marked a record object it contains as created, updated, or deleted.

This method always returns `false` if you execute it immediately after invoking any one of the following methods on the JSDO:

- `fill( )`
- `saveChanges( )`, if the `autoApplyChanges` property is also set to `true`
- `acceptChanges( )`
- `rejectChanges( )`

A typical use of this method is to determine if there are any changes in JSDO memory that you want to save to local storage using the JSDO `saveLocal( )` method.

Example

The following code fragment shows an example of how you might use `hasChanges( )` to decide how to save JSDO memory to local storage:

```
dataSet = new progress.data.JSDO( 'dsStaticData' );
dataSet.fill();
dataSet.autoApplyChanges = false;

/* Work done with the dataSet JSDO memory */
.
.
.
if (dataSet.hasChanges())
{
  dataSet.saveLocal(progress.data.JSDO.CHANGES_ONLY);
}
else
{
  dataSet.saveLocal(progress.data.JSDO.ALL_DATA);
}
```

The fragment first invokes the `fill( )` method on a JSDO (`dataSet`) to load JSDO memory, sets the `autoApplyChanges` property on the JSDO to not automatically accept or reject changes saved to the server based on the success or failure of the save, and after a certain amount of work is done with the JSDO, decides to save **all** the data in JSDO memory to the default local storage area, or to save **only** the pending changes, based on whether any pending changes currently exist in JSDO memory.

See also:

acceptChanges( ) method on page 44, autoApplyChanges property on page 82, fill( ) method on page 97, hasData( ) method on page 127, rejectChanges( ) method on page 165, saveChanges( ) method on page 174, saveLocal( ) method on page 188

hasData( ) method

Returns `true` if record objects can be found in any of the tables referenced in JSDO memory (with or without pending changes), or in only the single table referenced on the JSDO, depending on how the method is called; and returns `false` if no record objects are found in either case.

Return type: `boolean`

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Syntax

```
jsdo-ref.hasData ( )
jsdo-ref.table-ref.hasData ( )
```

jsdo-ref

A reference to the JSDO. If you call the method on *jsdo-ref*, the method verifies if any data is available in the JSDO, whether it is created for a single-table or a multi-table resource.

table-ref

A table reference on the JSDO. If you call the method on *table-ref*, the method verifies if any data is available in the referenced table. If the `useRelationships` property is `true`, this includes related data in any other JSDO table(s) with which the referenced table has a parent-child relationship in the JSDO.

This method always returns `true` immediately after the `fill( )` method successfully loads JSDO memory with one or more record objects.

Three cases where this method returns `false` include when executing this method:

1. After a JSDO is instantiated but before `fill( )` or any other method (such as `addRecords( )`) has been invoked to load its JSDO memory with records
2. After the `fill( )` method completes successfully, but returns no records, possibly because none match the specification of its *filter* parameter
3. After the `saveChanges( )` method completes successfully on a JSDO, where its `autoApplyChanges` property set to `true` and all the records in the specified JSDO, or its table reference, were marked for deletion

Two typical uses of this method include determining if there is any data in JSDO memory that you might want to save in local storage using the JSDO `saveLocal( )` method, or that you might not want to lose by replacing JSDO memory, using the JSDO `readLocal( )` method, with other data previously saved in local storage.

Example

The following code fragment shows an example of how you might use `hasData( )` to decide when to save JSDO memory to local storage:

```
dataSet = new progress.data.JSDO( 'dsStaticData' );
dataSet.fill();

/* Work done with the dataSet JSDO memory */
.
.
.
if (dataSet.hasData())
{
  dataSet.saveLocal();
}
```

The fragment first invokes the `fill( )` method on a JSDO (`dataSet`) to load JSDO memory, and after a certain amount of work is done with the JSDO, decides to save all the data in JSDO memory to the default local storage area when it finds that records exist in JSDO memory to save.

See also:

addRecords( ) method on page 61, fill( ) method on page 97, hasChanges( ) method on page 126, saveChanges( ) method on page 174, readLocal( ) method on page 164, saveLocal( ) method on page 188

invalidate( ) method

Terminates the login session managed by the current `JSDOSession` object and permanently disables the object, rendering it unable to start a new login session.

This method throws an error if the underlying communication layer throws an error.

On a successful invalidate, the `JSDOSession` object sets its `connected` property to `false`. If the invalidate fails to logout the session because it cannot access the server, the object still sets its `connected` property to `false` because it is no longer available to manage any login session.

Once `invalidate( )` is executed, no further communication can occur between the mobile app and the server using this `JSDOSession` object. Thus, any call made to its `addCatalog( )`, `isAuthorized( )`, `login( )`, or `ping( )` method, or any attempt to instantiate a `progress.data.JSDO` associated with this `JSDOSession` object, throws an exception. In addition, any calls to methods on an existing `JSDO` instance that communicate with the server associated with this `JSDOSession` throw an exception as well.

This method is always executed asynchronously and returns results in callbacks that you register using methods of a Promise object returned as the method value.

Note: In order to invoke this method successfully, jQuery Promises must be supported in your development environment. Otherwise, the method throws an exception.

Return type: jQuery Promise

Applies to: progress.data.JSDOSession class on page 25

Syntax

```
invalidate ( )
```

Promise method signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that `invalidate( )` returns:

Syntax:

```
promise.done( function ( session , result , info ) )
promise.fail( function ( session , result , info ) )
promise.always( function ( session , result , info ) )
```

promise

A reference to the Promise object that is returned as the value of the `invalidate( )` method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDOSession class on page 25.

session

A reference to the `JSDOSession` object on which `invalidate( )` was called.

result

A constant indicating the overall result of the call that can have one of the following values:

- **`progress.data.Session.SUCCESS`** — The invalidate of the `JSDOSession` object completed successfully.
- **`progress.data.Session.GENERAL_FAILURE`** — The invalidate failed because of some error.

Note: It is not always necessary to test the value of *`result`* in a Promise method callback for the `invalidate( )` method, especially if the callback is registered using *`promise`*`.done( )` or *`promise`*`.fail( )`, where the callback always executes with the same value (`progress.data.Session.SUCCESS` and `progress.data.Session.GENERAL_FAILURE`, respectively).

---

info

A JavaScript object that can have the following properties:

- **`errorObject`** — An error object thrown while attempting to terminate the login session.
- **`xhr`** — A reference to the XMLHttpRequest object sent to the web server to terminate the login session.

Example

The following code fragment calls the `invalidate( )` method on the `JSDOsession` object, `session`. This example uses `try` and `catch` blocks in case `invalidate( )` throws an unexpected exception, and displays messages returned from Promise handlers accordingly:

```
var serviceURI = "http://oemobiledemo.progress.com/OEMobileDemoServicesForm/",
session = new progress.data.JSDOSession({
            "authenticationModel": progress.data.Session.AUTH_TYPE_FORM,
            "serviceURI": serviceURI
    });

try { // Login, add catalogs, create additional JSDO instances,
      // and do JSDO tasks for this session, ...then(...) ...
    session.invalidate()
    .then(function(session, result, info) {
        // Success handler of invalidate
        return "Employee session invalidated successfully";
    }, function(session, result, info) {
        // Failure handler of invalidate
        if ( result === progress.data.Session.GENERAL_FAILURE ) {
            if ( info.errorObject ) {
                // Process error object thrown during invalidate...
            }
            if ( info.xhr ) {
                // Process XHR sent during invalidate...
            }
            return "Employee session invalidate failed. Unspecified error.";
        }
        else {
            return "Unexpected failed invalidate result";
        }
    })
    .then(function (msg) {
        console.log(msg);
    });
}
catch(ex) {
    console.log("There was an unexpected error from attempted invalidate: " + ex);
}
```

See also:

invalidateAllSessions( ) stand-alone function on page 130, logout( ) method (JSDOSession class) on page 150, connected property on page 95

invalidateAllSessions( ) stand-alone function

A stand-alone function that invalidates all current `progress.data.JSDOSession` instances that were created and initialized using the `progress.data.getSession( )` stand-alone function.

This function combines the features of:

- Finding all existing `JSDOSession` instances created using the `getSession( )` stand-alone function.
- Calling the `invalidate( )` method on each instance, which in turn calls the `logout( )` method on the instance.

After `invalidateAllSessions( )` executes, any call made to the `addCatalog( )`, `isAuthorized( )`, `login( )`, or `ping( )` method on an existing `JSDOSession` instance, or any attempt to instantiate a `progress.data.JSDO`, throws an exception. In addition, calls to any method on an existing `JSDO` instance that communicates with the server associated with an invalidated `JSDOSession` throws an exception as well.

This method is always executed asynchronously and returns results in callbacks that you register using methods of a Promise object returned as the function value.

Note: In order to invoke this function successfully, jQuery Promises must be supported in your development environment. Otherwise, the function throws an exception.

Note: This function does not support proxy servers (servers that function as a security service).

Return type: jQuery Promise

Applies to: The `progress.data` namespace

Syntax

```
progress.data.invalidateAllSessions ( )
```

Promise method signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the function that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that `invalidateAllSessions( )` returns:

Syntax:

```
promise.done( function ( result , info ) )
promise.fail( function ( result , info ) )
promise.always( function ( result , info ) )
```

Note: The `always( )` method is always passed what the `done( )` or the `fail( )` methods are passed.

promise

A reference to the Promise object that is returned as the value of the `invalidateAllSessions( )` function. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDOSession class on page 25.

result

An `integer` constant indicating the overall result of the call that can have one of the following values:

- **`progress.data.Session.SUCCESS`** — Invalidate of all `JSDOSession` instances has completed successfully.
- **`progress.data.Session.GENERAL_FAILURE`** — The invalidate failed because of some error.

Note: It is not always necessary to test the value of *result* in a Promise method callback for `invalidateAllSessions( )`, especially if the callback is registered using *promise*`.done( )`, where the callback always executes with the same value (`progress.data.Session.SUCCESS`).

info

A JavaScript object that can have the following properties:

- **`errorObject`** — An error object thrown while attempting to terminate any `JSDOSession` instance.
- **`xhr`** — A reference to the XMLHttpRequest object sent to the web server for the `JSDOSession` object that threw the error object.

Example

The following code fragment calls the `invalidateAllSessions( )` method to terminate and invalidate the login session for every current `JSDOSession` instance that is created using `progress.data.getSession( )`. This example uses `try` and `catch` blocks in case `invalidateAllSessions( )` throws an unexpected exception, and displays messages returned from Promise handlers accordingly:

```
var serviceURI = "http://oemobiledemo.progress.com/OEMobileDemoServicesForm/",
catalogURI =
"http://oemobiledemo.progress.com/OEMobileDemoServicesForm/static/CustomerService.json";

// create a JSDOSession for a specified web application
// that requires Form authentication
try {
    // create and initialize login session
    progress.data.getSession( {
            authenticationModel : progress.data.Session.AUTH_TYPE_FORM,
            serviceURI : serviceURI,
            catalogURI : catalogURI,
            username: "formuser",
            password: "formpassword"
    })
    .then(function( session, result ) {

        // Create a JSDO and fill it with data
        var jsdoCustomer = new progress.data.JSDO('Customer');
        return jsdoCustomer.fill()

    }, function(result, info) {
        console.log("getSession failed");
    })
    .then(function (jsdo, result, request) {
        console.log("Fill went okay!");
        // Do other JSDO related tasks here
    }, function (jsdo, result, request) {
        console.log("Not filling it today");
        // Do JSDO related failure stuff here
    })
    .always(function () {
        // By the time we get here, we should be done with all JSDO related stuff
        // and cleaning up
        return progress.data.invalidateAllSessions();
    })
    .then(function( result, info ) {
        console.log("All sessions are invalidated!");
    }, function() {
        console.log("A session failed validation!");
    });
} catch (ex) {
    console.log("Exception: " + ex.message);
}
```

Note that this example creates only a single login session. Any additional login sessions can be created, and associated JSDO tasks performed, anywhere appropriate in the same Promise chain prior to the Promise `always( )` method handler where `invalidateAllSessions( )` is invoked.

See also:

invalidate( ) method on page 128, logout( ) method (JSDOSession class) on page 150, getSession( ) stand-alone function on page 121, progress.data.JSDOSession class on page 25

invocation method

A custom method on the JSDO that executes an Invoke operation defined for a Data Object resource.

For an OpenEdge resource, this invocation method can be implemented as a corresponding ABL routine on an OpenEdge application server. This implementation can be any ABL routine that is defined in its Data Service Catalog with an Invoke operation type. The invocation method name can be the same as the ABL routine or an alias, as defined by the resource. The method passes any ABL input parameters as properties of an object parameter. The method returns results from the ABL routine, including any return value and output parameters, as properties of a request object that is returned by the method.

This method also causes an `offline` or `online` event to fire if it detects that there has been a change in the JSDO login session's online state.

This custom invocation method can be executed either synchronously, returning results in an object return value, or asynchronously, returning similar results in an `afterInvoke` event callback, depending on a parameter that you pass. You can also call the invocation method asynchronously using the JSDO `invoke( )` method, which returns results as a jQuery Promise. For more information on using `invoke( )`, see the description of the invoke( ) method on page 136.

Note: The results of an Invoke operation have no effect on JSDO memory. However, you can use the JSDO `addRecords( )` method to merge any record data that is returned by an invocation method into JSDO memory.

Return type: request object on page 37 or `undefined`

Applies to: progress.data.JSDO class on page 12

Syntax

```
op-name ( [ input-object ] [ , async-flag ] )
```

op-name

The name (specified as an identifier) of the invocation method as defined by the resource.

input-object

An object whose properties and values match the case-sensitive names and data types of the input parameters specified for the server routine that implements the invocation method. If the implementing routine does not take input parameters, specify `null` or leave out the argument entirely.

async-flag

A `boolean` that when `true` causes the method to execute asynchronously and when `false` causes the method to execute synchronously. The default value is `true`.

Returning and handling results

For a synchronous invocation, the method returns, as its value, a request object that contains several properties depending on the status of the Invoke operation.

For an asynchronous invocation, the method return value is `undefined`, but returns a similar request object as input to any event handler (callback) function subscribed to the following named events that fire in the following operational order:

1. beforeInvoke event on page 89
2. afterInvoke event on page 70

For a successful execution, if there are any output parameters or a method return value, they are returned as properties of an object referenced by the `response` property of the request object. For an OpenEdge resource, the referenced properties for output parameters match the case-sensitive names and data types of the output parameters defined for the implementing ABL routine. Any method return value is returned by a JSDO-defined `_retVal` property with a matching JavaScript data type. For more information on data type mappings for an OpenEdge resource, see OpenEdge ABL to JavaScript data type mappings on page 212.

For an unsuccessful execution in an OpenEdge resource, the request object `response` property references an object that can contain a both a `_retVal` property, which is a `string` value with the error information, and an `_errors` property, which is an array with information on one or more errors. For more information on error information returned in the response property, see the description of the response property on page 172.

Note: For an OpenEdge resource, if you are calling an invocation method that either sends a single table object or a ProDataSet object (with one or more tables) as a property of *`input-object`* or returns a single table object or ProDataSet object as a property of the `response` property object, you need to apply a rule in order to access this table or ProDataSet object. The rule is that wherever you de-reference or reference a table or ProDataSet object, you must reference that value twice, separated by a period or a colon, depending on the context. The reason is that the table or ProDataSet name is both the name of the parameter defined for the ABL routine and also the name of the JavaScript object containing the JSON data returned from the server. For example, to access a table object, `ttCust` returned by the `response` property in a `request` object, you must code the following de-reference: `request.response.ttCust.ttCust`. Similarly, if you pass `ttCust` to an invocation method, `InputTT( )`, you must code the following reference: `jsdo.InputTT( {ttCust: {ttCust:ttCust}});`

Note: For an OpenEdge resource, if the invocation method passes ProDataSet as an input or output parameter, that ProDataSet can contain before-image data. However, the invocation method does no processing of the before-image data in any way. You must therefore manage the object appropriately. For example, you can use an output ProDataSet containing before-image data as a *`merge-object`* parameter to the `addRecords( )` method as long as your JSDO is created with the same ProDataSet schema and the OpenEdge resource supports before-imaging.

Example

The following code fragment shows an invocation method (`getOrderTotalAndStatus( )`) called asynchronously on a JSDO as defined for an OpenEdge resource (`dsCustomerOrder`), with results returned in the callback (`onAfterInvokeGetOrderTotalAndStatus`) subscribed to the `afterInvoke` event:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );
dataSet.subscribe( 'afterInvoke', 'getOrderTotalAndStatus',
                   onAfterInvokeGetOrderTotalAndStatus );

dataSet.getOrderTotalAndStatus( paramObject ); // An asynchronous call is the default

function onAfterInvokeGetOrderTotalAndStatus( jsdo , success , request )
    if (success) {

        var response = request.response;
        var ordTotal = response._retVal;
        var ordStatus = response.pcStatus;
        /* process successful results */
        . . .

    }
    else {
        if (request.response && request.response._errors &&
            request.response._errors.length > 0) {

            var lenErrors = request.response._errors.length;
            for (var idxError=0; idxError < lenErrors; idxError++) {

                var errorEntry = request.response._errors[idxError];
                var errorMsg = errorEntry._errorMsg;
                var errorNum = errorEntry._errorNum;
                /* handle error */
                . . .
            }
        }
    }

};
```

Note that the callback is subscribed only for execution of `getOrderTotalAndStatus( )` and the call to `getOrderTotalAndStatus( )` is asynchronous by default, with its input parameters passed in `paramObject`. For a successful execution in the callback, the order total is returned as the method return value in the JSDO-defined `_retVal` property, and the one output parameter with the order status is returned as the value of the ABL-defined `pcStatus` property, both of which are returned in the `response` property object. For an unsuccessful execution, the callback inspects the `response` property to return one or more error messages in an array of objects.

See also:

addRecords( ) method on page 61, invoke( ) method on page 136, fill( ) method on page 97, record property on page 165, request object on page 37, saveChanges( ) method on page 174

invoke( ) method

Asynchronously calls a custom invocation method on the JSDO to execute an Invoke operation defined by a Data Object resource.

OpenEdge resources can define and implement custom invocation methods that you can call using this method.

The asynchronous execution of the specified invocation method using `invoke( )` returns results using a jQuery Promise. You can also directly call an invocation method on the JSDO either synchronously, returning results as its return value, or asynchronously, returning results using a callback subscribed to the `afterInvoke` event.

Note: In order to call `invoke( )` successfully, jQuery Promises must be supported in your development environment. Otherwise, the method throws an exception.

For more information on the possible implementations for custom invocation methods, and how to call them directly on the JSDO, see the description of the invocation method on page 134.

Note: The results of an Invoke operation have no effect on JSDO memory. However, you can use the JSDO `addRecords( )` method to merge any record data that is returned by an invocation method into JSDO memory.

Return type: jQuery Promise

Applies to: progress.data.JSDO class on page 12

Syntax

```
invoke ( op-name [ , input-object ] )
```

op-name

A `string` that specifies the name of the invocation method as defined by the resource.

input-object

An object whose properties and values match the case-sensitive names and data types of the input parameters specified for the server routine that implements the invocation method. If the implementing routine does not take input parameters, specify `null` or leave out the argument entirely.

Promise method callback signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that `invoke( )` returns:

Syntax:

```
promise.done( function ( jsdo , success , request ) )
promise.fail( function ( jsdo , success , request ) )
promise.always( function ( jsdo , success , request ) )
```

promise

A reference to the Promise object that is returned as the value of the `invoke( )` method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDO class on page 12.

jsdo

A reference to the JSDO that invoked the `invoke( )` method (Invoke operation) on its resource. For more information, see the description of the jsdo property on page 143 of the request object.

success

A `boolean` that is `true` if the Invoke operation was successful. For more information, see the description of the success property on page 202 of the request object.

Note: It is not always necessary to test the value of *`success`* in a Promise method callback for the `invoke( )` method, especially if the callback is registered using *`promise`*`.done( )` and *`promise`*`.fail( )`, where the callback executes according to this value.

---

request

A reference to the request object returned after the `invoke( )` method completed execution and returned any results from its Invoke operation on the server. For more information, see the description of the request object on page 37.

Returning and handling results

This method returns results asynchronously in two different ways, and in the following order, depending on the development environment used to build the mobile app:

1. **Using JSDO named events for all environments** — The following events fire before and after the `invoke( )` method executes, respectively, with results handled by callback functions that you subscribe as documented for each event:
   a. beforeInvoke event on page 89
   b. afterInvoke event on page 70
2. **Using a Promise object returned for environments that support jQuery Promises** — Any callbacks that you register using Promise object methods all execute both **after** the `invoke( )` method itself and **after** any subscribed `afterInvoke` event callbacks for the *`op-name`*-specified invocation method complete execution. Note that the signatures of all Promise method callbacks match the signature of the `afterInvoke` event callback function so you can specify an existing `afterInvoke` event callback as the callback function that you register using any Promise method.

Because the callbacks that you register with any returned Promise methods execute only after all subscribed `afterInvoke` event callbacks for the *`op-name`*-specified invocation method complete execution, you can invoke logic in the Promise method callbacks to modify any processing done by the event callbacks.

If the `invoke( )` method completes successfully, the *`success`* parameter for any `afterInvoke` event callback or Promise method and the `success` property of each handler's *`request`* parameter object are both set to `true`. Otherwise, both the *`success`* parameter and `success` property are set to `false`, and you can read any error results by inspecting the setting of the `response` property in the same *`request`* parameter object.

The request object returned as an input parameter to any subscribed event callback or to any registered Promise object callback contains several properties depending on the status of the Invoke operation.

For a successful execution, if there are any output parameters or a method return value, they are returned as properties of an object referenced by the `response` property of the request object. For an OpenEdge resource, the referenced properties for output parameters match the case-sensitive names and data types of the output parameters defined for the implementing ABL routine. Any method return value is returned by a JSDO-defined `_retVal` property with a matching JavaScript data type. For information on data type mappings for an OpenEdge resource, see OpenEdge ABL to JavaScript data type mappings on page 212.

For an unsuccessful execution in an OpenEdge resource, the request object `response` property references an object that can contain a both a `_retVal` property, which is a `string` value with the error information, and an `_errors` property, which is an array with information on one or more errors. For more information on error information returned in the response property, see the description of the response property on page 172.

For more information on working with the results of invocation methods that you call with `invoke( )`, see the notes on handling results in the description of the invocation method on page 134.

Example

The following code fragment shows `invoke( )` calling an invocation method (`getOrderTotalAndStatus( )`) as defined for an OpenEdge resource (`dsCustomerOrder`), with asynchronous results returned in appropriate callbacks registered by methods of the returned Promise object:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );

dataSet.invoke( "getOrderTotalAndStatus" , paramObject ).done(
    function( jsdo, success, request ) {

        var response = request.response;
        var ordTotal = response._retVal;
        var ordStatus = response.pcStatus;
        /* process successful results */
        . . .

    }).fail(
    function( jsdo, success, request ) {
        if (request.response && request.response._errors &&
            request.response._errors.length > 0) {

            var lenErrors = request.response._errors.length;
            for (var idxError=0; idxError < lenErrors; idxError++) {

                var errorEntry = request.response._errors[idxError];
                var errorMsg = errorEntry._errorMsg;
                var errorNum = errorEntry._errorNum;
                /* handle error */
                . . .
            }
        }
    });
```

Note that for the call to `getOrderTotalAndStatus( )` in the previous example, its input parameters are passed in `paramObject`, as required by the implementing ABL routine. Depending on the success of the `getOrderTotalAndStatus( )` call, the appropriate callback registered for the Promise executes, making it unnecessary to test its `success` parameter.

Thus, when the callback registered by the Promise's `done( )` method executes for a successful `getOrderTotalAndStatus( )` call, the order total is returned as the method return value in the JSDO-defined `_retVal` property, and the one output parameter with the order status is returned as the value of the ABL-defined `pcStatus` property, both of which are returned in the `response` property object. When the callback registered by the Promise's `fail( )` method executes for an unsuccessful `getOrderTotalAndStatus( )` call, the callback inspects the `response` property to return one or more possible error messages in an array of objects.

See also:

addRecords( ) method on page 61, fill( ) method on page 97, invocation method on page 134, record property on page 165, request object on page 37, saveChanges( ) method on page 174

isAuthorized( ) method

Determines if the current JSDOSession object has authorized access to the web application specified by its serviceURI property setting.

This method also causes an offline or online event to fire if it detects that there has been a change in the JSDOSession object's online state.

This method is always executed asynchronously and returns results in callbacks that you register using methods of a Promise object returned as the method value.

Return type: jQuery Promise

Applies to: progress.data.JSDOSession class on page 25

Syntax

```
isAuthorized ( )
```

Promise method signatures

jQuery Promise objects define methods that you can call to register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that isAuthorized( ) returns:

Syntax:

```
promise.done( function ( session , result , info ) )
promise.fail( function ( session , result , info ) )
promise.always( function ( session , result , info ) )
```

promise

A reference to the Promise object that is returned as the value of the isAuthorized( ) method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDOSession class on page 25.

session

A reference to the JSDOSession object on which isAuthorized( ) was called.

result

One of the following string constant values depending on the JSDOSession authorization or online status identified by isAuthorized( ):

- **progress.data.Session.SUCCESS** — An authentication appropriate to the JSDOSession object's authenticationModel setting previously succeeded and is currently sufficient to give access to the web application specified by its serviceURI setting.
- **progress.data.Session.AUTHENTICATION_FAILURE** — An authentication appropriate to the JSDOSession object's authenticationModel setting previously succeeded, but the

JSDOSession is no longer authorized to access its web application, for example, because the login session, established using HTTP Form authentication, has expired.

- **progress.data.Session.LOGIN_AUTHENTICATION_REQUIRED** — No successful authentication was completed for this JSDOSession object (there was no login session), or if there was a login session established, the JSDOSession has terminated its access to its web application by logging out.

  Note: This value is also returned when invoking the method after a page refresh if page refresh support is not enabled for the JSDOSession. For more information, see Additional information on page 141.

- **progress.data.Session.GENERAL_FAILURE** — An error resulted from trying to contact its web application. You might find more information by consulting the *info* callback parameter.

The authorization status (*result* parameter value) is determined by whether the JSDOSession object can contact its web application to see if it is allowed access. If a login session was never previously authenticated, the JSDOSession never even tries to contact its web application. This communication also depends on the authenticationModel setting passed to the object's constructor.

info

A JavaScript object that can have the following properties:

- **xhr** — A reference to the XMLHttpRequest object, if any, sent to the web server to make the authorization request to the web application. If no request was made, this property is undefined.
- **offlineReason** — A string constant that is set only if isAuthorized( ) determines that the JSDOsession object is disconnected from its web application or its associated application server. The returned value indicates the reason for the JSDOsession offline state. Possible values include some of those returned by a call to the ping( ) method, as follows:
  - **progress.data.Session.DEVICE_OFFLINE** — The device itself is offline. For example, it might be in airplane mode, or it might be unable to pick up a Wi-Fi or cell signal.
  - **progress.data.Session.SERVER_OFFLINE** — The web server is not available. For a Rollbase Data Object Service, this is the web server for the public or private cloud. For an OpenEdge Data Object Service, this is the Tomcat Java servlet container.
  - **progress.data.Session.WEB_APPLICATION_OFFLINE** — The server is running, but the Java web application that implements the Data Object Service is not deployed.

Additional information

The behavior of isAuthorized( ) in a web app after a browser page refresh depends on whether page refresh support is enabled by passing an operative value for the name property in the JSDOSession object constructor.

For example, if page refresh support has **not** been enabled, regardless of the authentication model, after the user initiates a page refresh, but before a **new** login session has been established by calling the login( ) method, the callback *result* parameter is always set to progress.data.Session.LOGIN_AUTHENTICATION_REQUIRED. For more information on page refresh support, see the description of the progress.data.JSDOSession class on page 25 constructor.

Note: If the JSDOSession authentication model is HTTP Basic, page refresh support is never enabled.

Example

The following code fragment shows how you can use `isAuthorized( )` to check the online state of a given `JSDOSession` object with page refresh support enabled in the `JSDOSession` constructor:

```
$(document).ready(function(){
    if (!ablSession) {
        // Create a session if it does not yet exist
        var ablSession = new progress.data.JSDOSession(
             {  serviceURI: "http://localhost:8810/FormFCS",
                authenticationModel: progress.data.Session.AUTH_TYPE_FORM,
                name: "ablSessionKey"   // enable page refresh support
             } );
    }

    // Check whether there had been a login and we are still authorized
    ablSession.isAuthorized()
    .done( function(session, result, info) {
        startApp(); // start web app
    })
    .fail( function(session, result, info) {
        switch (result) {
            case progress.data.Session.LOGIN_AUTHENTICATION_REQUIRED:
            case progress.data.Session.AUTHENTICATION_FAILURE:
                showLogin(); // login, then start web app
                break;
            case progress.data.Session.GENERAL_FAILURE:
                // possibly some sort of offline problem
                if (info.offlineReason) {
                    alert("Offline: " + info.offlineReason);
                }
                break;
        }
    });
```

This web app example initially verifies if the instance variable (`ablSession`) for a `JSDOSession` already references an instance and creates the instance if it does not already exist, with HTTP Form authentication and page refresh enabled.

It then immediately calls `isAuthorized( )` to determine if an existing login session is already established from a restored prior session state. If it is established and authorized to access its web application, the `done( )` callback executes calling `startApp()`, which is a method that continues web app execution using the restored JSDO login session. If it is not established or authorized (likely for a first-time execution or prior logout), the `fail( )` callback executes.

If the `fail( )` callback executes either because initial authentication for the `JSDOSession` has not been done or authorization for a restored `JSDOSession` failed, it calls `showLogin()`, which is a method that prompts the user for credentials to call the `login( )` method on the `JSDOSession` and then perhaps call `startApp()` (or the equivalent) if the new session login succeeds.

If the `fail( )` callback executes for some other reason, it displays any available offline status (`info.offlineReason`) that might be the cause of the failure. Although not shown, it can also inspect the returned XMLHttpRequest object (`info.xhr`) for more information on the failure.

See also:

name property (JSDOSession class) on page 154, login( ) method (JSDOSession class) on page 145, offline event on page 155, online event on page 157

jsdo property

An object reference to the JSDO that performed the operation returning the request object.

Data type: progress.data.JSDO class on page 12

Access: Read-only

Applies to: request object on page 37

The `jsdo` property is available for all JSDO events. This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe or register to JSDO events, any returned jQuery Promises, and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO. This object is also returned as the value of any JSDO invocation method that you execute synchronously.

See also:

fill( ) method on page 97, invocation method on page 134, invoke( ) method on page 136, saveChanges( ) method on page 174

JSDOs property

Returns an array of JSDOs that use the current `JSDOSession` object to communicate with their Data Object Services.

Data type: JSDO array

Access: Read-only

Applies to: progress.data.JSDOSession class on page 25

See also:

pingInterval property on page 163, services property on page 190

jsrecord property

A reference to the record object that was created, updated, or deleted by the current record-change operation.

Data type: progress.data.JSRecord class on page 35

Access: Read-only

Applies to: request object on page 37

The `jsrecord` property is available only for the following JSDO events:

- `afterCreate`
- `afterDelete`
- `afterUpdate`

- `beforeCreate`
- `beforeDelete`
- `beforeUpdate`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe both to JSDO events and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO.

See also:

saveChanges( ) method on page 174

jsrecords property

An array of references to record objects that are created, updated, or deleted on the server for a Submit operation (invoking `saveChanges(true)`) on a Data Object resource that supports before-imaging.

Data type: Array of progress.data.JSRecord class on page 35

Access: Read-only

Applies to: request object on page 37

For a before-image resource, such as an OpenEdge ProDataSet resource, if the record change on a given `JSRecord` object referenced in this array has executed with an error, the `data` property object of this record object contains a property named `"prods:rowState"` that has one of the following `string` values indicating the type of record change that was applied to the record:

- `"created"`
- `"modified"`
- `"deleted"`

To access this property (with a special character (`:`) in its name), you must reference it as an index reference on the `data` property object, as in this example, where `request` is the request object returned from the Submit operation:

```
for (var i = 0;i < request.jsrecords.length;i++) {
    var jsrecord = request.jsrecords[i];
    var jsrecError = jsrecord.getErrorString();
    if (jsrecError) {
        /* handle record-change error */
        console.log("Record change: "
                    + jsrecord.data["prods:rowState"]);
        console.log("Error: " + jsrecError);
    }
}
```

Note: The `"prods:rowState"` property is always available in this `data` property object **only** if the JSDO `autoApplyChanges` property is `false` when you invoke `saveChanges(true)`.

The `jsrecords` property is available only for the following JSDO events or in the request object returned to a jQuery Promise callback, and **only** after invoking `saveChanges(true)` on the JSDO:

- `afterSaveChanges`
- `beforeSaveChanges`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe both to JSDO events and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO.

See also:

saveChanges( ) method on page 174

login( ) method (JSDOSession class)

Note: The recommended approach is to use the `getSession( )` method.

Starts a JSDO login session in a web application for the current `JSDOSession` object by sending an HTTP request with specified user credentials to the web application URI specified in the object's constructor.

This method throws an error if the method arguments are invalid or if the underlying communication layer throws an error.

On a successful login, the `JSDOSession` object sets its `connected` property to `true`. If the login fails, the object leaves its `connected` property set to `false`.

This method is always executed asynchronously and returns results in callbacks that you register using methods of a Promise object returned as the method value.

Note: In order to invoke this method successfully, jQuery Promises must be supported in your development environment. Otherwise, the method throws an exception.

Note: Before invoking this method, ensure that you set the `authenticationModel` configuration property in the constructor of the `JSDOSession` object correctly (see the notes on authentication models).

Note: If the browser or mobile device has already authenticated a JSDO login session, this method completes successfully.

Note: This method does not support proxy servers (servers that function as a security service).

Note: As a recommended alternative to using this method on an existing `JSDOSession` object, you can create a new, initialized `JSDOSession` using the `progress.data.getSession( )` stand-alone function. In one call, this function instantiates the object, invokes its `login( )` method with specified credentials, then invokes its `addCatalog( )` method to load a specified Data Service Catalog. For more information, see the getSession( ) stand-alone function on page 121 description.

Return type: jQuery Promise

Applies to: progress.data.JSDOSession class on page 25

Syntax

```
login ( [username , password][ , parameter-object] )
```

username

A string expression containing a user ID for the method to send to the web server for authentication.

Note: The userName property of the JSDOSession object returns the most recent value passed to this method for the current JSDOSession object.

password

A string expression containing a password for the method to send to the web server to authenticate the specified user.

parameter-object

An object that contains the following optional property:

- **iOSBasicAuthTimeout** — A number that specifies the time, in milliseconds, that the login( ) method waits for a response before returning an error. This error might mean that the user entered invalid credentials. If you set this value to zero (0), no timeout is set, and login( ) can wait indefinitely before returning an error. If you do not set the iOSBasicAuthTimeout property, login( ) uses 4000 (4 seconds) as the default value.

  Note: Any non-zero timeout value (default or otherwise) for this *parameter-object* property operates **only** under certain conditions. Otherwise, any setting of this property has no effect. For more information, see the notes of the progress.data.JSDOSession class on page 25 description.

Note: The login( ) method on the progress.data.Session class relies on the class constructor to specify the web application URI and always uses the default web resource, /static/home.html, as the login target.

Promise method signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that login( ) returns:

Syntax:

```
promise.done( function ( session , result , info ) )
promise.fail( function ( session , result , info ) )
promise.always( function ( session , result , info ) )
```

promise

A reference to the Promise object that is returned as the value of the `login( )` method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDOSession class on page 25.

session

A reference to the `JSDOSession` object on which `login( )` was called.

result

A constant indicating the overall result of the call that can have one of the following values:

- **`progress.data.Session.AUTHENTICATION_SUCCESS`** — The JSDO login session started successfully. You can use JSDOs to access any Data Object Services supported by the web application to which the `JSDOSession` object has logged in.
- **`progress.data.Session.AUTHENTICATION_FAILURE`** — JSDO login failed because of invalid user credentials (*`username`* or *`password`*).
- **`progress.data.Session.GENERAL_FAILURE`** — JSDO login failed because of a non-authentication failure.

  Note: This value can also be returned if invalid user credentials triggered a login timeout according to the value of the *`parameter-object`*`.iOSBasicAuthTimeout` property.

Note: It is not always necessary to test the value of *`result`* in a Promise method callback for the `login( )` method, especially if the callback is registered using *`promise`*`.done( )`, where the callback always executes with the same value (`progress.data.Session.AUTHENTICATION_SUCCESS`).

info

A JavaScript object that can have the following properties:

- **`errorObject`** — Any error object thrown as a result of sending a login request to the web server.

  Note: If this object is thrown because of a login timeout triggered according to the value of the *`parameter-object`*`.iOSBasicAuthTimeout` property, the `message` property of the error object is set to `"login timeout expired"`.

- **`xhr`** — A reference to the XMLHttpRequest object sent to the web server to start a JSDO login session.

You can also return the result of the most recent login attempt on the current `JSDOSession` object by reading its `loginResult` property. For a more specific status code returned in the HTTP response, you can also check the value of the `loginHttpStatus` property.

Note: You can log out from a web application and then log in again using the same `JSDOSession` object. The login will use the same `serviceURI` and `authenticationModel` settings originally passed to the constructor, but you must pass any required credentials each time `login( )` is called.

General web server interaction

The general web server interaction with and response to this method depends on the authentication model that the web server uses and how resources are accessed and protected. You configure the authentication model for each web application deployed to the Apache Tomcat and specify both the web application URI and its corresponding authentication model to the `JSDOSession` object constructor. For more information on the authentication models that a `JSDOSession` object supports, see the description of the constructor for the progress.data.JSDOSession class on page 25.

For more information on these authentication models and how to configure them for a web application, see the sections on configuring web server authentication models in the server documentation for your Data Object Service.

Caution: You must be sure that security is configured to complete authentication before the application requests resources in the Data Service Catalog. Although it is possible to configure application security so that only the Data Object resources in the Catalog require authentication, Progress does not recommend this approach. Instead, Progress recommends that you require authentication for application resources in addition to those defined in the Catalog, and require that the authentication occur prior to accessing any resources in the Catalog. Once the user is authenticated, the web server provides access to all other resources, including Catalog resources, according to the user's authorization settings.

Note: Unless the application design guarantees that the user will be prompted by the web browser or native device container to provide credentials before a `login( )` call occurs, Progress recommends (in some cases requires) that the mobile or web app pass the credentials as parameters to the `login( )` method. In addition, you must correctly pass the value of the `authenticationModel` configuration property to the `JSDOSession` object's constructor. Coding the mobile or web app in this way ensures that the proper credentials are submitted to the server and promotes a favorable user experience.

Example

The following code fragment calls the `login( )` method on the `JSDOsession` object, `empSession`. This example uses the callbacks registered by the Promise `done( )` and `fail( )` methods to check the result of the call along with any error object thrown as a result of the request, then assembles an appropriate message to display in an alert box. It also uses a `try-catch` block in case the `login( )` method throws an error object instead of sending the request to the server:

```
var msg;
                  var xhr;

                  try {
                  empSession.login( userName, passWord ).done(
                  function( session, result, info ) {
                  msg = "Logged in successfully”;
                  }).fail(
                  function( session, result, info ) {
                  if ( result
                  === progress.data.Session.AUTHENTICATION_FAILURE ) {
                  msg = "Employee Login failed. Authentication error";
                  }
                  else if ( result
                  === progress.data.Session.GENERAL_FAILURE  ) {
                  msg = "Employee Login failed. Unspecified error";
                  if ( info.errorObject ) {
                  // Process error object thrown during login . . .
                  }
                  if ( info.xhr ) {
                  // Process XHR sent during login . . .
                  }
                  }
                  xhr = info.xhr;
                  });
                  }
                  catch(errObj) {
                  msg = "Employee Login failed. Error attempting to call login";
                  msg = msg + '\n' + errObj.message;
                  }

                  msg = msg +
                  "\nloginResult: " + empSession.loginResult +
                  "\nloginHttpStatus: " +  empSession.loginHttpStatus +
                  "\nuserName: " +  empSession.userName +
                  "\nLogin XHR: " +  xhr;

                  alert(msg);
```

See also:

addCatalog( ) method (JSDOSession class) on page 50, authenticationModel property (JSDOSession class) on page 81, connected property on page 95, getSession( ) stand-alone function on page 121, loginHttpStatus property on page 149, loginResult property on page 150, logout( ) method (JSDOSession class) on page 150, offline event on page 155, online event on page 157, serviceURI property on page 191, userName property on page 209

loginHttpStatus property

Returns the specific HTTP status code returned in the response from the most recent login attempt on the current `JSDOSession` object.

Data type: `number`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 25

See also:

login( ) method (JSDOSession class) on page 145

loginResult property

Returns the return value of the `login( )` method, which is the basic result code for the most recent login attempt on the current `JSDOSession` object.

Data type: `number`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 25

Possible `loginResult` values include the following numeric constants:

- **`progress.data.Session.LOGIN_SUCCESS`** — JSDO login session started successfully.
- **`progress.data.Session.LOGIN_AUTHENTICATION_FAILURE`** — JSDO login failed because of invalid user credentials.
- **`progress.data.Session.LOGIN_GENERAL_FAILURE`** — JSDO login failed because of a non-authentication failure.

For a more specific status code returned in the HTTP response, you can check the value of the `loginHttpStatus` property.

The value of this property is `null` prior to the first login attempt and after a successful logout (until the next login attempt).

See also:

login( ) method (JSDOSession class) on page 145, loginHttpStatus property on page 149

logout( ) method (JSDOSession class)

Terminates the login session on the web application managed by the current `JSDOSession` object and leaves the object available to start a new login session with a call to its `login( )` method.

This method throws an error if the underlying communication layer throws an error.

On a successful logout, the `JSDOSession` object sets its `connected` property to `false`. If the logout fails, the object leaves its `connected` property set to `true`, unless the failure happened because the app cannot access the server.

Once `logout( )` is executed, no further communication (other than a `login( )` call) can occur between the mobile app and the server using this `JSDOSession` object. However, any catalogs loaded in the object remain available to create and maintain JSDOs, though these JSDOs cannot make requests to the server until a new login session is established for the object (including for anonymous access).

This method is always executed asynchronously and returns results in callbacks that you register using methods of a Promise object returned as the method value.

Note: In order to invoke this method successfully, jQuery Promises must be supported in your development environment. Otherwise, the method throws an exception.

Note: If you do not need to reuse the current `JSDOSession` instance, you can call `invalidate( )` (instead of `logout( )`) to permanently disable the instance and prevent any additional calls to its `login( )` method. If you want to permanently disable all current `JSDOSession` instances, you can also call `invalidateAllSessions( )` instead.

Return type: jQuery Promise

Applies to: progress.data.JSDOSession class on page 25

Syntax

```
logout ( )
```

Promise method signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that `logout( )` returns:

Syntax:

```
promise.done( function ( session , result , info ) )
promise.fail( function ( session , result , info ) )
promise.always( function ( session , result , info ) )
```

promise

A reference to the Promise object that is returned as the value of the `logout( )` method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDOSession class on page 25.

session

A reference to the `JSDOSession` object on which `logout( )` was called.

result

A constant indicating the overall result of the call that can have one of the following values:

- **`progress.data.Session.SUCCESS`** — The session logout completed successfully.
- **`progress.data.Session.GENERAL_FAILURE`** — The session logout failed because of some error.

Note: It is not always necessary to test the value of *result* in a Promise method callback for the logout( ) method, especially if the callback is registered using *promise*.done( ) or *promise*.fail( ), where the callback always executes with the same value (progress.data.Session.SUCCESS and progress.data.Session.GENERAL_FAILURE, respectively).

info

A JavaScript object that can have the following properties:

- **errorObject** — An error object thrown while attempting to terminate the login session.
- **xhr** — A reference to the XMLHttpRequest object sent to the web server to terminate the login session.

Detailed logout behavior

When this method terminates the associated login session, the JSDOSession object can be re-used to start a new session using the same serviceURI and authenticationModel configuration settings originally passed to the object's constructor. The JSDOSession object's properties retain their values from the previous login session, with the following exceptions:

- clientContextId is reset to null.
- loginHttpStatus is reset to null.
- loginResult is reset to null.
- userName is reset to null.

Existing JSDOs and catalog information are not affected by a successful execution of logout( ). However, any attempt to call addCatalog( ) or a JSDO method that requires contacting the server results in an error object being thrown.

Example

The following code fragment calls the `logout( )` method on the `JSDOsession` object, `session`. This example uses `try` and `catch` blocks in case `logout( )` throws an unexpected exception, and displays messages returned from Promise handlers accordingly:

```
var serviceURI = "http://oemobiledemo.progress.com/OEMobileDemoServicesForm/",
session = new progress.data.JSDOSession({
            "authenticationModel": progress.data.Session.AUTH_TYPE_FORM,
            "serviceURI": serviceURI
    });

try { // Login, add catalogs, create additional JSDO instances,
      // and do JSDO tasks for this session, ...then(...) ...
    session.logout()
    .then(function(session, result, info) {
        // Success handler of logout
        return "Employee session logged out successfully";
    }, function(session, result, info) {
        // Failure handler of logout
        if ( result === progress.data.Session.GENERAL_FAILURE ) {
            if ( info.errorObject ) {
                // Process error object thrown during logout...
            }
            if ( info.xhr ) {
                // Process XHR sent during logout...
            }
            return "Employee session logout failed. Unspecified error.";
        }
        else {
            return "Unexpected failed logout result";
        }
    })
    .then(function (msg) {
        console.log(msg);
    });
}
catch(ex) {
    console.log("There was an unexpected error from attempted logout: " + ex);
}
```

See also:

invalidate( ) method on page 128, invalidateAllSessions( ) stand-alone function on page 130, login( ) method (JSDOSession class) on page 145, offline event on page 155, online event on page 157

name property (JSDO class)

Returns the name of the Data Object resource for which the current JSDO was created.

This value must match the name of a resource provided by the Data Object Service for which a login session has already been started.

Note: To set this property, you must pass its value to the JSDO constructor.

Data type: `string`

Access: Read-only

Applies to: progress.data.JSDO class on page 12

See also:

services property on page 190

name property (JSDOSession class)

Returns the value of any `name` property to enable page refresh support that was passed to the constructor of the current `JSDOSession` object.

Data type: `string`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 25

See the description of the `JSDOSession` class constructor for more information on the setting and use of this property value.

See also:

getSession( ) stand-alone function on page 121, isAuthorized( ) method on page 140

objParam property

A reference to the object, if any, that was passed as an input parameter to the JSDO method that has returned the current request object.

If no object parameter was passed to the method, the `objParam` property is `undefined`.

Data type: `Object`

Access: Read-only

Applies to: request object on page 37

The `objParam` property is available for all JSDO events where an object parameter is passed to the JSDO method, including `saveChanges(true)`, where such object parameters are passed using a single Submit operation, or in the request object returned to a jQuery Promise callback.

However, this property **does not** apply to the following events:

- `afterSaveChanges`, when `saveChanges( )` or `saveChanges(false)` is called
- `beforeCreate`
- `beforeDelete`
- `beforeUpdate`
- `beforeSaveChanges`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe or register to JSDO events, any returned jQuery Promises, and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO. This object is also returned as the value of any JSDO invocation method that you execute synchronously.

See also:

fill( ) method on page 97, invocation method on page 134, invoke( ) method on page 136, saveChanges( ) method on page 174

offline event

Fires when the current `JSDOSession` object detects that the device on which it is running has gone offline, or that the web application to which it has been connected is no longer available.

This event always fires when the device on which the `JSDOSession` object is created goes offline (that is, the device is disconnected from the network). For this event to fire when the web application to which it has been connected is no longer available, the `JSDOSession` object must have previously:

1. Been connected to the web application using the object's `login( )` method
2. Not been disconnected from the web application using the object's `logout( )` method

Applies to: progress.data.JSDOSession class on page 25

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdosession , off-line-reason , request )
```

session

A reference to the `JSDOSession` object that has detected the offline condition.

off-line-reason

A `string` constant indicating the reason that the offline event has been fired. Possible constants include:

- **`progress.data.Session.APPSERVER_OFFLINE`** — (OpenEdge Data Object Services only) The other components necessary to run the service are available, but the associated OpenEdge application server is offline.
- **`progress.data.Session.DEVICE_OFFLINE`** — The device itself is offline. For example, it might be in airplane mode, or it might be unable to pick up a Wi-Fi or cell signal.
- **`progress.data.Session.SERVER_OFFLINE`** — The web server is not available. For a Data Object Service, this is the web server for the public or private cloud. For an OpenEdge Data Object Service, this is the Tomcat Java servlet container.
- **`progress.data.Session.WEB_APPLICATION_OFFLINE`** — The server is running, but the Java web application that implements the Data Object Service is not deployed.

Note: The `progress.data.Session.DEVICE_OFFLINE` condition will fire the `offline` event on a `JSDOSession` object even if the object has not yet authenticated.

You can use the string values of these constants directly to display or log messages, or test the values only and respond to the result in some other way.

request

If the offline condition was detected as a result of a request sent on behalf of a JSDO, this is a reference to the `request` object used to make the request. For more information, see the description of the request object on page 37. If the offline event is the result of the device going offline or because of a call to the `ping( )` method (either directly or because the `pingInterval` property is set greater than 0), this parameter is `null`.

Note: If the underlying database accessed by the OpenEdge application server is unavailable, this will not cause the `offline` event to fire, nor will the fact that a particular Data Object Service contained by the hosting web application was not loaded when the application started.

Example

The following code fragment subscribes the function, `onSessionOffline`, to handle the `offline` event fired on the session, `mySession`:

```
/* subscribe to the offline event */
                  var mySession = new progress.data.JSDOSession(
                  { serviceURI: "http://localhost/CustService",
                  authenticationModel: progress.data.Session.AUTH_TYPE_FORM
                  };
                  mySession.subscribe('offline', onSessionOffline );

                  /* some code that might cause mySession to detect that the
                  application is offline */

                  . . .

                  function onSessionOffline( session , offlineReason , request ) {

                  document.write(offLineReason);
                    switch( offlineReason ) {
                  case progress.data.Session.APPSERVER_OFFLINE:
                  document.write("Check with the server system administrator");
                  . . .
                  break;
                  case progress.data.Session.DEVICE_OFFLINE:
                  document.write("Ensure that your device is online");
                  . . .
                  break;
                  case progress.data.Session.SERVER_OFFLINE:
                  document.write("Check with the server system administrator");
                  . . .
                  break;
                  case progress.data.Session.WEB_APPLICATION_OFFLINE:
                  document.write("Check with the server system administrator");
                  . . .
                  break;
                  default:
                  document.write("Check with the server system administrator");
                  };

                  };
```

See also:

connected property on page 95, login( ) method (JSDOSession class) on page 145, logout( ) method (JSDOSession class) on page 150, online event on page 157, ping( ) method (JSDOSession class) on page 160, subscribe( ) method (JSDOSession class) on page 202, unsubscribe( ) method (JSDOSession class) on page 207

online event

Fires when the current JSDOSession object detects that the device on which it is running has gone online after it was previously offline, or that the web application to which it is connected is now available after it was previously unavailable.

This event always fires when the device on which the JSDOSession object is created goes online after having been offline (that is, the device is reconnected to the network). For this event to fire when the web application to which it has been connected is now available after having been unavailable, the JSDOSession object must have previously:

1. Been instantiated using getSession( )
2. Not been disconnected from the web application using the object's logout( ) method
3. Detected an offline event as a result of sending a Data Object Service request to the web application or executing the ping( ) method

Applies to: progress.data.JSDOSession class on page 25

The following parameters appear in the signature of any event handler (callback) function (or functions) that you subscribe to this event:

Syntax

```
function ( jsdosession , request )
```

session

A reference to the JSDOSession object that has detected the online condition.

request

If the online condition was detected as a result of a request sent on behalf of a JSDO, this is a reference to the request object used to make the request. For more information, see the description of the request object on page 37. If the online event is the result of the device itself coming back online or because of a call to the ping( ) method (either directly or because the pingInterval property is set greater than 0), this parameter is null.

This event can fire because:

- The device on which the app is running was previously offline (in airplane mode or could not connect to a Wi-Fi network), but is now online again.

  Note: This condition will fire the online event on a JSDOSession object.

- A JSDOSession object that previously fired its offline event because of a failed attempt to contact a Data Object Service has now been used successfully to contact that same Data Object Service.

Example

The following code fragment subscribes the function, `onSessionOnline`, to handle the `online` event fired on the session, `mySession`:

```
/* subscribe to the online event */
                  var mySession = new progress.data.JSDOSession(
                  { serviceURI: "http://localhost/CustService",
                  authenticationModel: progress.data.Session.AUTH_TYPE_FORM
                  };
                  mySession.subscribe('online', onSessionOnline );

                  /* some code that might cause mySession to detect that the
                  application is online */

                  . . .

                  function onSessionOnline( session , request ) {

                  document.write("Your best mobile app is back to work! "
                  + "You might be prompted to login again.");

                  };
```

See also:

connected property on page 95, login( ) method (JSDOSession class) on page 145, logout( ) method (JSDOSession class) on page 150, offline event on page 155, ping( ) method (JSDOSession class) on page 160, subscribe( ) method (JSDOSession class) on page 202, unsubscribe( ) method (JSDOSession class) on page 207

onOpenRequest property

Returns the reference to a user-defined callback function that the `JSDOSession` object executes to modify a request object before sending the request object to the server.

For example, this function might add a message header by modifying the XMLHttpRequest object used to send the request.

You do not typically use this property, because the `JSDOSession` object properly handles preparation of the request object for normal circumstances.

Data type: `function`

Access: Readable/Writable

Applies to: progress.data.JSDOSession class on page 25

By default, the value of the `onOpenRequest` property is `null`, meaning that the request object is sent without modification. If the value is set to a callback function, the function takes a single object parameter.

Syntax

This is the syntax for setting this property to a function reference:

```
                  mySession.onOpenRequest = funcRef
```

mySession

A reference to the JSDOSession object for which the request object is to be modified before sending a request to the server.

funcRef

A reference to a JavaScript callback function that has the following signature:

Syntax:

```
function [func-name] ( param )
```

func-name

The optional name of a function you have defined external to the property assignment. Alternatively, you can specify *funcRef* as the entire inline function definition without *func-name*.

param

An Object that has the following properties:

- **xhr** — An object reference to the XMLHttpRequest object (XHR) used to send the request. The current request object can be modified by the function. When the callback is called, XMLHttpRequest.open( ) will already have been called on the XHR , but the callback can call open( ) again, overriding the effects of the first open( ) call. When the callback function is used for a addCatalog( ) or logout( ) call, although it should not be necessary and is not recommended, it is possible to replace the XHR entirely by creating a new object and assigning it as the value of the xhr property.
- **verb** — The HTTP operation (GET, PUT, etc.) to be performed by the request.
- **uri** — The URI to which the request is addressed.
- **session** — A reference to the Session object that invoked the callback.
- **async** — A boolean specifying whether the request is asynchronous (true) or synchronous (false).

Caution: For a JSDOSession object, if the callback function is used for a addCatalog( ) or logout( ) call, and if it calls XMLHttpRequest.open( ), the async property value passed to that open( ) call must be true (i.e., the open( ) method must specify that the request is to be sent asynchronously).

If you assign a callback function as the value of onOpenRequest, it remains in effect for all requests for the duration of the session unless it is replaced by another function or is set to null.

Example

Be sure to reset the value of the property as necessary, as in the following example for the `JSDOSession` object, `myJSDOsession`:

```
myJSDOsession.onOpenRequest = function( params ) {
  params.xhr.setRequestHeader('Authorization', auth);
};

myJSDOsession.login(username, password).done(
function(JSDOsession, result, info) {
JSDOsession.onOpenRequest = null;
. . .
}).fail(
function(JSDOsession, result, info) {
alert("Login failed");
});
```

See also:

request object on page 37, xhr property on page 209

ping( ) method (JSDOSession class)

Determines the online state of the current `JSDOSession` object from its ability to access the web application that it manages, and for an OpenEdge web application, from detecting if its associated application server is running.

This method also causes an `offline` or `online` event to fire if the ping detects that there has been a change in the `JSDOSession` object's online state.

This method throws an exception if the `JSDOSession` object has not logged into its web application or has since logged out.

This method is always executed asynchronously and returns results in callbacks that you register using methods of a Promise object returned as the method value.

Return type: jQuery Promise

Applies to: progress.data.JSDOSession class on page 25

Syntax

```
ping ( )
```

Promise method signatures

jQuery Promise objects define methods that you can call to register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that `ping( )` returns:

Syntax:

```
promise.done( function ( session , result , info ) )
promise.fail( function ( session , result , info ) )
promise.always( function ( session , result , info ) )
```

promise

A reference to the Promise object that is returned as the value of the `ping( )` method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDOSession class on page 25.

session

A reference to the `JSDOSession` object on which `ping( )` was called.

result

A `boolean` that indicates the online state of the `JSDOSession` object on which `ping( )` is called. If set to `true`, `ping( )` has determined that the current `JSDOSession` object is connected and logged into a web application (and its associated application server, for an OpenEdge Data Object Service), or if set to `false`, the object is disconnected from any web application (**or** its associated application server, for an OpenEdge Data Object Service).

info

A JavaScript object that can have the following properties:

- **xhr** — A reference to the XMLHttpRequest object sent to the web server to make the ping request to the web application.
- **offlineReason** — A `string` constant that `ping( )` sets only if it determines that the `JSDOsession` object is disconnected from any web application or its associated application server. The constant value indicates the reason for its offline state. Possible values are the same as those that can be passed to the `offline` event callback function as the value of its *off-line-reason* parameter, and include:
  - **progress.data.Session.APPSERVER_OFFLINE** — (OpenEdge Data Object Services only) The other components necessary to run the service are available, but the associated OpenEdge application server is offline.
  - **progress.data.Session.DEVICE_OFFLINE** — The device itself is offline. For example, it might be in airplane mode, or it might be unable to pick up a Wi-Fi or cell signal.
  - **progress.data.Session.SERVER_OFFLINE** — The web server is not available. For a Rollbase Data Object Service, this is the web server for the public or private cloud. For an OpenEdge Data Object Service, this is the Tomcat Java servlet container.
  - **progress.data.Session.WEB_APPLICATION_OFFLINE** — The server is running, but the Java web application that implements the Data Object Service is not deployed.

Detailed ping behavior

A `JSDOSession` object is considered to be in an online state if all of the following are true:

- The object can communicate with its web application. That is, the web server is running, the web application is started, and it has accessible Data Object Services.

- For OpenEdge Data Object Services only, the associated OpenEdge application server that the web application accesses is running and accessible.
- The `ping()` service defined in the Progress Application Server for OpenEdge catalog is available.

When you execute `ping( )` on a `JSDOSession` instance, the method indicates the instance's current online status through its returned Promise object using the most commonly registered callbacks as follows:

- If the instance is online, the returned Promise object invokes any callback you register using the object's `done( )` method. The `done( )` callback's `result` parameter is always set to `true`.
- If the instance is offline, the returned Promise object invokes any callback you register using the object's `fail( )` method. The `fail( )` callback's `result` parameter is always set to `false`.
- The returned Promise object invokes any callback you register using the object's `always( )` method with the value of the callback's `result` parameter indicating the instance's online status as described for the Promise method signatures above.

Note: You can have `ping( )` executed automatically by setting the `pingInterval` property on the `JSDOSession` object, which specifies the time interval between invocations of `ping( )`. However, automatic invocations do not return Promise objects, but only fire `offline` and `online` events appropriately when a change in the `JSDOSession` instance's online status is detected.

Note: For OpenEdge Data Object Services, an OpenEdge application server supports `ping( )` using an OpenEdge-defined ABL class, `OpenEdge.Rest.Admin.AppServerStatus`. This class responds to a REST ping service call to its `ServerStatus( )` method, which indicates that the application server is available when the method returns successfully. You can also define a version of this method in your own user-defined ABL class that returns a custom string value when `ping( )` returns successfully, and you can retrieve this value from the `xhr` object reference returned by `ping( )`. For more information, see the sections on constructing and debugging REST requests in the administration documentation for the Progress Application Server for OpenEdge.

Example

The following code fragment shows how you can use `ping( )` to check the online state of a given `JSDOSession` object after logging in:

```
var mySession = new progress.data.JSDOSession ( { . . . } );

/* These session events can fire on device online
   status changes even prior to logging in */
mySession.subscribe('offline', onSessionOffline );
mySession.subscribe('online', onSessionOnline );

. . .

mySession.login().done( // Anonymous login
    function( session, result, info ) {

        . . .

        ping().done(
            function( session, result, info ) {
                console.log("Session ping result: Online");
                }
            }).fail(
            function( session, result, info ) {
                console.log("Session ping result: Offline -- "
                            + info.offlineReason);
                // Process info.xhr, if necessary, for more information
            });

        . . .

    }).fail( /* Process failed login . . . */ );

. . .

function onSessionOffline( pSession , pOfflineReason , pRequest ) {
  console.log("The session status has changed to: Offline -- "
              + pOfflineReason);
};

function onSessionOnline( pSession , pRequest ) {
  console.log("The session status has changed to: Online");
};
```

See also:

connected property on page 95, offline event on page 155, online event on page 157, pingInterval property on page 163

pingInterval property

A number that specifies the duration, in milliseconds, between one automatic execution of the current `JSDOSession` object's `ping( )` method and the next.

Setting this property to a value greater than zero (0) causes the `JSDOSession` object to begin executing its `ping( )` method, and when execution completes, to repeatedly execute the method after the specified delay. If you set its value to zero (0), no further execution of `ping( )` occurs after any current execution completes. The default value is zero (0).

Data type: `number`

Access: Readable/Writable

Applies to: progress.data.JSDOSession class on page 25

You can set `pingInterval` to start the automatic execution of `ping( )` any time after you create the `JSDOSession` object. However, `ping( )` does not begin executing until and unless you have successfully instantiated a `JSDOSession` via `getSession( )`.

Note that when you call the `ping( )` method directly, you have several options for how to call it to get the results. You do not have these options and you cannot get results directly from each automatic execution of `ping( )` that begins from a setting of `pingInterval`. The effects from this automatic execution are limited to causing the `JSDOSession` or `Session` object to fire its `offline` or `online` event, and to change the value of its `connected` property, when a given `ping( )` execution detects a change in the object's online status.

See also:

connected property on page 95, JSDOs property on page 143, login( ) method (JSDOSession class) on page 145, offline event on page 155, online event on page 157, ping( ) method (JSDOSession class) on page 160

readLocal( ) method

Clears out the data in JSDO memory and replaces it with all the data stored in a specified local storage area, including any pending changes and before-image data, if they exist.

Return type: `boolean`

Applies to: progress.data.JSDO class on page 12

Working record: After execution, no working records are set for the tables of the JSDO.

Syntax

```
readLocal ( [ storage-name ] )
```

`storage-name`

The name of the local storage area whose data is to replace the data in JSDO memory. If *`storage-name`* is not specified, blank, or `null`, the name of the default storage area is used. The name of this default area is `jsdo_`*`serviceName`*`_`*`resourceName`*, where *`serviceName`* is the name of the Data Object Service that supports the JSDO instance, and *`resourceName`* is the name of the resource (table, dataset, etc.) for which the JSDO instance is created.

This method returns `true` if it successfully reads the data from the local storage area; it then replaces JSDO memory with this data. If the storage area has no data (is empty), this clears JSDO memory instead of replacing it with any data, and the method also returns `true`. If *`storage-name`* does not exist, but otherwise encounters no errors, the method ignores (does not clear) JSDO memory and returns `false`. If the method does encounter errors (for example, with the data in the specified storage area), it also leaves JSDO memory unchanged and throws an exception.

You can call the JSDO `saveChanges( )`, `acceptChanges( )`, or `rejectChanges( )` method after calling this method, and any changes read into JSDO memory from local storage are handled appropriately.

Example

The following code fragment replaces the data in JSDO memory with all the data currently stored in the default storage area:

```
dataSet = new progress.data.JSDO( 'dsStaticData' );
dataSet.fill();
dataSet.saveLocal();
.
.
.
dataSet.readLocal();
```

See also:

acceptChanges( ) method on page 44, fill( ) method on page 97, rejectChanges( ) method on page 165, saveChanges( ) method on page 174, saveLocal( ) method on page 188

record property

A property on a JSDO table reference that references a `JSRecord` object with the data for the working record of a table referenced in JSDO memory.

If no working record is set for the referenced table, this property is `undefined`.

Data type: progress.data.JSRecord class on page 35

Access: Read-only

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

The table reference that provides this property can either be the value of a property on the JSDO with the name of a referenced table in JSDO memory or a reference to the JSDO itself if the JSDO references only a single table.

The field values and any additional properties for the working record are provided by the object returned by the `data` property in the `JSRecord` object that is returned by the `record` property.

See also:

data property on page 96

rejectChanges( ) method

Rejects changes to the data in JSDO memory for the specified table reference or for all table references of the specified JSDO.

If the method succeeds, it returns `true`. Otherwise, it returns `false`.

Note: This method is most useful when the JSDO `autoApplyChanges` property is set to `false`. In this case, you typically invoke this method **after** calling the `saveChanges( )` method in order to cancel a series of changes that have failed on the server. If the `autoApplyChanges` property is `true`, the JSDO automatically accepts or rejects changes for the specified table reference, or for all table references of the specified JSDO, based on the success of the corresponding record-change operations on the server resource.

Note: Rejecting all pending changes in JSDO memory—or even pending changes for a single table reference—because only some were unsuccessful on the server might be too broad an action for your application. If so, consider using `rejectRowChanges( )` to reject changes a single table record at a time. For more information, see the description of `rejectRowChanges( )` method.

Return type: `boolean`

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Working record: The working record is set depending on the changes rejected.

Syntax

```
jsdo-ref.rejectChanges ( )
jsdo-ref.table-ref.rejectChanges ( )
```

jsdo-ref

A reference to the JSDO. If you call the method on *jsdo-ref*, the method rejects changes for all table references in the JSDO.

table-ref

A table reference on the JSDO. If you call the method on *table-ref*, the method rejects changes for the specified table reference.

When you reject changes on a table reference, this method backs out all pending changes to the record objects in the specified table in JSDO memory, and uses any before-image data to return each record to its original data values before the pending changes were made. When you reject changes on the JSDO reference, the method backs out all pending changes to the record objects in all tables of the JSDO, and uses any before-image data to return each record to its original data values before the pending changes were made. As the specified changes are rejected, the method also empties any associated before-image data, clears all associated error message settings, and removes the associated pending record change indications from JSDO memory.

Note: Regardless if you call `rejectChanges( )`, any error message settings that result from Data Object resource operations invoked by the most recent execution of the `fill( )` or `saveChanges( )` method remain available for return by the `getErrors( )` method until the next execution of either `fill( )` or `saveChanges( )`. For more information, see the getErrors( ) method on page 110 description.

This method is especially useful for a resource that supports before-imaging (such as an OpenEdge ProDataSet) and handles the results of all record changes sent using a Submit operation as part of a single server transaction that undoes all the record changes in response to any single record-change error. When the Submit operation returns with an error, calling this method ensures that JSDO memory is synchronized with all the record changes that were undone as part of the server transaction.

Note: After this method rejects changes, and if you have set up automatic sorting using the `autoSort` property, all the record objects for affected table references are sorted accordingly. If the sorting is done using sort fields, any `string` fields are compared according to the value of the `caseSensitive` property.

Caution: If you have already successfully applied these changes on the server using the `saveChanges( )` method, **do not** invoke this method if you want the affected client data to be consistent with the corresponding data on the server.

Example

The following code fragment shows a JSDO created so it **does not** automatically accept or reject changes to data in JSDO memory after a call to the `saveChanges( )` method. Instead, it subscribes a handler for the JSDO `afterSaveChanges` event to determine if all changes to the `eCustomer` table in JSDO memory should be accepted or rejected based on the success of all resource Create, Update, and Delete operations on the server. To change the data for a record, a jQuery event is also defined on an update button to update the corresponding `eCustomer` record in JSDO memory with the current field values entered in a customer detail form (`#custdetail`):

```
dataSet = new progress.data.JSDO( { name: 'dsCustomerOrder',
                                                  autoApplyChanges : false } );
dataSet.eCustomer.subscribe('afterSaveChanges', onAfterSaveCustomers, this);

$('#btnUpdate').bind('click', function(event) {
  var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
  if (jsrecord) {jsrecord.assign();};
});

// Similar controls might be defined to delete and create eCustomer records...

$('#btnSave').bind('click', function(event) {
  dataSet.saveChanges();
});

function onAfterSaveCustomers(jsdo, success, request) {
  if (success)  {
    jsdo.eCustomer.acceptChanges();
    // Additional actions associated with accepting the pending changes...
  }
  else  {
    jsdo.eCustomer.rejectChanges();
    // Additional actions associated with rejecting the pending changes...
  }
}
```

When the update button is clicked, the event handler uses the `findById( )` method to find the original record (`jsrecord`) with the matching internal record ID (`#id`) and invokes the `assign( )` method on `jsrecord` with an empty parameter list to update its fields in `eCustomer` with any new values entered into the form. You might define similar events and controls to delete `eCustomer` records and add new `eCustomer` records.

An additional jQuery event also defines a save button that when clicked invokes the `saveChanges( )` method to apply all pending changes in JSDO memory to the server. After the method completes, and all results have been returned to the client from the server, the JSDO `afterSaveChanges` event fires, and if any resource operations on the server were **not** successful, the handler calls `rejectChanges( )` to reject all pending `eCustomer` changes in JSDO memory.

Note: This example shows the default invocation of `saveChanges( )`, which invokes each resource record-change operation, one record at a time, across the network. For a resource that supports before-imaging (such as an OpenEdge ProDataSet), you can also have `saveChanges( )` send all pending record change operations across the network in a single Submit operation. For more information and an example, see the description of the `saveChanges( )` method.

See also:

acceptChanges( ) method on page 44, autoApplyChanges property on page 82, autoSort property on page 82, caseSensitive property on page 93, rejectRowChanges( ) method on page 168, saveChanges( ) method on page 174

rejectRowChanges( ) method

Rejects changes to the data in JSDO memory for a specified record object.

This can be the working record of a table reference or the record specified by a `JSRecord` object reference. If the method succeeds, it returns `true`. Otherwise, it returns `false`.

Note: This method is most useful when the JSDO `autoApplyChanges` property is set to `false`. In this case, you might invoke this method in the callback for the corresponding JSDO event fired in response to an unsuccessful record-change operation on the server resource that was invoked by executing the `saveChanges( )` method. If the `autoApplyChanges` property is `true`, the JSDO automatically accepts or rejects changes to the record object based on the success of the corresponding resource operation on the server.

Return type: `boolean`

Applies to: progress.data.JSRecord class on page 35, progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Working record: The working record is set depending on the changes rejected.

Syntax

```
jsrecord-ref.rejectRowChanges ( )
jsdo-ref.rejectRowChanges ( )
jsdo-ref.table-ref.rejectRowChanges ( )
```

jsrecord-ref

A reference to a `JSRecord` object for a table reference in JSDO memory.

You can obtain a `JSRecord` object by:

- Invoking a JSDO method that returns record objects from a JSDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a JSDO table reference that already has a working record.
- Accessing the *record* parameter passed to the callback of a JSDO `afterCreate`, `afterDelete`, or `afterDelete` event.

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference, and that table reference has a working record.

table-ref

A table reference on the JSDO that has a working record.

When you reject changes on a specified record object, this method makes the record reflect all pending changes to it in JSDO memory. As the specified changes are rejected, the method also empties any associated before-image data, clears any associated error message setting, and removes the associated pending change indications from JSDO memory.

Note: Regardless if you call `rejectRowChanges( )`, any error message settings that result from Data Object resource operations invoked by the most recent execution of the `fill( )` or `saveChanges( )` method remain available for return by the `getErrors( )` method until the next execution of either `fill( )` or `saveChanges( )`. For more information, see the getErrors( ) method on page 110 description.

Note: After this method rejects changes on a record, and if you have set up automatic sorting using the `autoSort` property, all the record objects for the affected table reference are sorted accordingly. If the sorting is done using sort fields, any `string` fields are compared according to the value of the `caseSensitive` property.

Caution: If you have successfully applied these JSDO changes to the server using the `saveChanges( )` method, **do not** invoke this method if you want the affected client data to be consistent with the corresponding data on the server.

Example

The following code fragment shows a JSDO created so it **does not** automatically accept or reject changes to data in JSDO memory after a call to the `saveChanges( )` method. Instead, it subscribes a single handler for each of the `afterDelete`, `afterCreate`, and `afterUpdate`, events to determine if changes to any `eCustomer` table record in JSDO memory should be accepted or rejected based on the success of the corresponding resource operation on the server. To change the data for a record, a jQuery event is also defined on a save button to update the corresponding `eCustomer` record in JSDO memory with the current field values entered in a customer detail form (`#custdetail`):

```
dataSet = new progress.data.JSDO( { name: 'dsCustomerOrder',
                                                  autoApplyChanges : false } );
dataSet.eCustomer.subscribe('afterDelete', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterCreate', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterUpdate', onAfterCustomerChange, this);

$('#btnSave').bind('click', function(event) {
  var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
  if (jsrecord) {jsrecord.assign();};
  dataSet.saveChanges();
});

// Similar controls might be defined to delete and create eCustomer records...

function onAfterCustomerChange(jsdo, record, success, request) {
  if (success)  {
    record.acceptRowChanges();
  }
  else
  {
    record.rejectRowChanges();
    // Perform other actions associated with rejecting this record change
  }
}
```

When the button is clicked, the event handler uses the `findById( )` method to find the original record with the matching internal record ID (`#id`) and invokes the `assign( )` method on `jsrecord` with an empty parameter list to update its fields in `eCustomer` with any new values entered into the form. It then calls the `saveChanges( )` method to invoke the resource Update operation to apply these record changes to the server. You might define similar events and controls to delete the `eCustomer` record or add a new `eCustomer` record.

After each resource operation for a changed `eCustomer` record completes, results of the operation are returned to the client from the server, and the appropriate event fires. If the operation was **not** successful, the handler calls `rejectRowChanges( )` to reject the record change associated with the event in JSDO memory. An advantage of using an event to manually reject a record change is that you can perform other actions associated with rejecting this particular change, such as displaying an alert to the user that identifies the error that caused the rejection.

Note: This example shows the default invocation of `saveChanges( )`, which invokes each resource record-change operation, one record at a time, across the network. For a resource that supports before-imaging (such as an OpenEdge ProDataSet), you can also have `saveChanges( )` send all pending record change operations across the network in a single Submit operation. For more information and an example, see the description of the `saveChanges( )` method.

See also:

acceptRowChanges( ) method on page 47, autoApplyChanges property on page 82, autoSort property on page 82, caseSensitive property on page 93, saveChanges( ) method on page 174, rejectChanges( ) method on page 165

remove( ) method

Deletes the specified table record referenced in JSDO memory.

The specified record can either be the working record of a referenced table or any record provided by a `JSRecord` object.

To synchronize the change on the server, call the `saveChanges( )` method.

Return type: `boolean`

Applies to: progress.data.JSRecord class on page 35, progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Working record: After execution, any prior working record setting for the affected table, and for any of its child tables, is no longer set.

Syntax

```
jsrecord-ref.remove ( )
jsdo-ref.remove ( )
jsdo-ref.table-ref.remove ( )
```

jsrecord-ref

A reference to a `JSRecord` object for a table record in JSDO memory.

You can obtain a `JSRecord` object by:

- Invoking a JSDO method that returns record objects from a JSDO table reference (`find( )`, `findById( )`, or `foreach( )`)
- Accessing the `record` property on a JSDO table reference that already has a working record.
- Accessing the *`record`* parameter passed to the callback of a JSDO `afterCreate`, `afterDelete`, or `afterDelete` event.
- Accessing each record object provided by the `jsrecords` property on the *`request`* object parameter passed to the callback of a JSDO `afterSaveChanges` event, or passed to the callback of any Promise object returned from the `saveChanges( )` method. The `jsrecords` property is only available on completion of a Submit operation (`saveChanges(true)`) on a resource that supports before-imaging, and **not** if the resource supports before-imaging without Submit.

jsdo-ref

A reference to the JSDO. You can call the method on *`jsdo-ref`* if the JSDO has only a single table reference, and that table reference has a working record.

table-ref

A table reference on the JSDO that has a working record.

Note: This method does not trigger an automatic sort and has no effect on any existing sort order established for the table reference. However, if there is a sort order that depends on the presence or absence of the record object you are removing, and you want to establish a new sort order with this record object absent, you must manually sort the remaining record objects using the `sort( )` method by passing it the same sort function that you used to establish the previous sort order.

Example

The following code fragment shows a jQuery event defined on a delete button to delete the record displayed in a customer detail form from the `eCustomer` table referenced in JSDO memory:

```
dataSet = new progress.data.JSDO( 'dsCustomerOrder' );

$('#btnDelete').bind('click', function(event) {
  var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
  if (jsrecord) {jsrecord.remove();};
  dataSet.saveChanges();
});
```

The form has been previously displayed with values from the same record. When the button is clicked, the event handler uses the `findById( )` method to find the original record with the matching internal record ID (`jsrecord`) and invokes the `remove( )` method on `jsrecord` to delete the record from `eCustomer`.

See also:

data property on page 96, saveChanges( ) method on page 174, sort( ) method on page 197

response property

Returns an object or `string` containing data and status information from an operation invoked on a Data Object resource.

Note: As detailed here, you can inspect the object or `string` value of this property to return certain types of error information from a resource operation, depending on the operation, the error origination, and whether or not the resource supports before-imaging. As of Progress Data Objects 4.3 and later, the JSDO provides a `getErrors( )` method that returns **all** types of error information for a specified table reference associated with a resource operation in a consistent manner, regardless of the operation, the error origination, and whether or not the resource supports before-imaging. For more information on this general error handling mechanism, see the getErrors( ) method on page 110 description.

Data type: `Object` or `string`

Access: Read-only

Applies to: request object on page 37

If a resource's Create, Read, Update, Delete (CRUD), or Submit operation returns successfully and the response is valid JSON that can be converted to a JavaScript object, the `response` property contains a reference to the resource's data that is returned from the server. If the server response is not valid JSON, the `response` property is `undefined`.

If a resource's Invoke operation returns successfully and has no return value or output parameters, the property is `null`. If the Invoke operation has a return value, you can read it as the value of the object's `_retVal` property. If the operation has output parameters, you can read these parameters as the values of object properties whose case-sensitive names and data types match the names and data types of the output parameters specified for the operation on the server.

If any CUD or Submit operation returns an error for an OpenEdge resource, the `response` property error object always contains the following properties for a resource that does **not** support before-imaging and **can** contain these properties for a resource that **supports** before-imaging:

- **`_retVal`** — A `string` with the value of any ABL `RETURN ERROR` string or `ReturnValue` property for a thrown `AppError` object
- **`_errors`** — An array of JavaScript objects, each of which contains two properties: `_errorMsg` with the ABL error message string and `_errorNum` with the error number, for one of possibly many ABL-returned errors

---

Note: This `response` property object can also return an error object with `_retVal` and `_errors` properties for an unhandled application server exception instead of an ABL application error.

If a given OpenEdge resource **does** support before-imaging, and the `response` property does not reference an error object for an unsuccessful result, information for an individual record-change errors (from CUD or Submit operations) is typically returned as part of the before-image data for each record object associated with the error, which you can return using the `getErrorString( )` method. For `after*` events of individual CUD operations, you can return this associated record object using the `jsrecord` property of the request object. In the callback for an `afterSaveChanges` event, or in any callback for a Promise object response to a Submit operation (`saveChanges(true)`) on a before-image resource, you can return the before-image data for every JSDO record object sent to the server using the `jsrecords` property of the request object.

Access to information on individual record-change error results depends on if the resource operation is a Create, Update, Delete (CUD), or a Submit operation and if the resource supports before-imaging. For more information, see the section on error handling in the description of the `saveChanges( )` method. See also the notes in this description of the `response` property.

The `response` property is available only for the following JSDO events or in the request object returned to a jQuery Promise callback:

- `afterCreate`
- `afterDelete`
- `afterFill`
- `afterInvoke`
- `afterSaveChanges`
- `afterUpdate`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe or register to JSDO events, any returned jQuery Promises, and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO. This object is also returned as the value of any JSDO invocation method that you execute synchronously.

Note: Error information returned with this property is most useful, and in one case is only possible, when the `autoApplyChanges` property is `false`, where you can individually reject or accept JSDO memory changes associated with each resource operation. When `autoApplyChanges` is `true`, the JSDO automatically rejects any JSDO memory changes associated with operation errors, but clears any error conditions and information associated with the operations only after the final `after*` event has been fired and handled (if handled) for the operation.

Note: Any any errors that result from a Data Object resource operation invoked by the most recent execution of the `fill( )` or `saveChanges( )` method remain available for return by the `getErrors( )` method until the next execution of `fill( )` or `saveChanges( )`, regardless of the setting of `autoApplyChanges`.

Note: If a given resource does not support before-imaging, any information for an individual record-change error from a CUD operation is typically returned in the `response` property, as described above. In the callback for an `afterSaveChanges` event, or in any callback for a Promise object, in response to executing `saveChanges(false)` (invoking CUD operations), you can also inspect the returned `response` property to find any error information, as described above.

See also:

autoApplyChanges property on page 82, fill( ) method on page 97, getErrors( ) method on page 110, getErrorString( ) method on page 115, invocation method on page 134, invoke( ) method on page 136, jsrecord property on page 143, jsrecords property on page 144, saveChanges( ) method on page 174, xhr property on page 209

saveChanges( ) method

Synchronizes to the server all record changes (creates, updates, and deletes) pending in JSDO memory for the current Data Object resource since the last call to the `fill( )` or `saveChanges( )` method, or since any prior changes have been otherwise accepted or rejected.

This method also causes an `offline` or `online` event to fire if it detects that there has been a change in the JSDO login session's online state.

This method always executes asynchronously and returns results (either or both) in subscribed JSDO event callbacks or in callbacks that you register using methods of a Promise object returned as the method value. A Promise object is always returned as the method value if jQuery Promises (or the exact equivalent) are supported in your development environment. Otherwise, this method returns `undefined`.

Return type: jQuery Promise or `undefined`

Applies to: progress.data.JSDO class on page 12

Working record: After execution, any prior working record settings for the tables of the JSDO are no longer set.

Syntax

```
saveChanges ( [ use-submit ] )
```

use-submit

An optional boolean parameter that when false (or not specified), tells saveChanges( ) to individually invoke all pending resource Create, Update, and Delete (CUD) operations, which are sent one record at a time across the network. You can use this option with a JSDO that accesses either a single-table or multi-table resource, with or without before-image support. Results for each record change are returned across the network to the JSDO after the operation on that record completes. The JSDO provides standard mechanisms to handle the results from each standard CUD operation (see the section on returning and handling results, below).

When true, this parameter tells saveChanges( ) to invoke a single Submit operation on the resource that handles any pending record-change (CUD) operations in a single network request. You can use this option only with a JSDO that accesses a single OpenEdge ProDataSet resource that supports before-imaging.

Note: With either type of operation (CUD or Submit) on an OpenEdge resource, the behavior of the operation on the server depends entirely on the Business Entity that implements the resource. To synchronize JSDO memory appropriately with the resource implementation, you must set the JSDO autoApplyChanges property and call the JSDO accept*Changes( ) and reject*Changes( ) as described later in this topic.

Promise method callback signatures

jQuery Promise objects define methods that register a callback function with a specific signature. The callback signatures depend on the method that returns the Promise. Following are the signatures of callbacks registered by methods in any Promise object that saveChanges( ) returns:

Syntax:

```
promise.done( function ( jsdo , success , request ) )
promise.fail( function ( jsdo , success , request ) )
promise.always( function ( jsdo , success , request ) )
```

promise

A reference to the Promise object that is returned as the value of the saveChanges( ) method. For more information on Promises, see the notes on Promises in the description of the progress.data.JSDO class on page 12.

jsdo

A reference to the JSDO that invoked the saveChanges( ) method (record-change operations) on its resource. For more information, see the description of the jsdo property on page 143 of the request object.

success

A boolean that is true if all record-change operations invoked on the resource by saveChanges( ) were successful. For more information, see the description of the success property on page 202 of the request object.

Note: It is not always necessary to test the value of *success* in a Promise method callback for the saveChanges( ) method, especially if the callback is registered using *promise*.done( ) and *promise*.fail( ), where the callback executes according to this value.

---

request

A reference to the request object returned after the `saveChanges( )` method completed execution and returned all results from its record-change operations on the server. For more information, see the description of the request object on page 37.

Default CUD operation execution: one changed record at a time

Without *use-submit*, the `saveChanges( )` method invokes the pending CUD operations in JSDO memory one record at a time across the network, and in the following general order of operation type:

1. **`"Delete"`** — All record deletions are applied.
2. **`"Create"`** — The creation of all new records is applied.
3. **`"Update"`** — Updates are applied to all modified records.

The sending of changes for multiple operations on the same record is optimized so the fewest possible changes are sent to the server. For example, if a record is first updated, then deleted in JSDO memory, only the deletion is sent to the server. However, note that all the changes to the record are applied to JSDO memory in a pending manner in case the deletion fails on the server.

If the JSDO is accessing an OpenEdge ProDataSet resource that is configured to support before-imaging, the JSDO also sends before-image data, along with each changed record, across the network to the server.

Note: Without *use-submit*, this method performs no batching of resource CUD operations. That is, the Delete operation is invoked over the network for each deleted record, followed by the Create operation for each created record, and finally by the Update operation for each updated record. So, even for a ProDataSet resource that supports before-imaging, each CUD operation executes over the network only one record at a time and cannot be part of a multi-record transaction.

Submit operation execution: multiple changed records at a time

If the JSDO is accessing an OpenEdge ProDataSet resource that supports before-imaging and the Submit operation, with *use-submit* specified as `true`, this method sends all pending Delete, Create, and Update record changes to the server in a single Submit operation. This Submit operation can thus manage a single, multi-record server transaction.

In this case, all the record changes, along with the before-image data, are sent in the ProDataSet to the server, to be processed by a single ABL routine that implements the Submit operation. This allows all associated server CUD operations to be part of a single server transaction in which all pending record changes are attempted, with all results returned in the ProDataSet in a single network response, including any error information for each record change.

If you specify *use-submit* as `true` for a ProDataSet resource, the implementing Business Entity must be created with before tables for every temp-table that the ProDataSet manages and configured to support both before-image data and invocation of the Submit operation. Otherwise, this method throws an exception.

If you specify *use-submit* as `true` for a temp-table resource, this method throws an exception.

Returning and handling results

This method returns results asynchronously in two different ways, and in the following order, depending on the development environment used to build the mobile app:

1. **Using JSDO named events for all environments** — The following events fire before and after the `saveChanges( )` method executes, respectively, with results handled by callback functions that you subscribe as documented for each event:

   a. beforeSaveChanges event on page 90
   b. beforeDelete event on page 87
   c. afterDelete event on page 67
   d. beforeCreate event on page 85
   e. afterCreate event on page 64
   f. beforeUpdate event on page 91
   g. afterUpdate event on page 76
   h. afterSaveChanges event on page 72

2. **Using a Promise object returned for environments that support jQuery Promises** — Any callbacks that you register using Promise object methods all execute both after the `saveChanges( )` method itself and after all subscribed JSDO `after*` event callbacks complete execution. Note that the signatures of all Promise method callbacks match the signature of the `afterSaveChanges` event callback function so you can specify an existing `afterSaveChanges` event callback as the callback function that you register using any Promise method.

Because the callbacks that you register with any returned Promise methods execute only after all subscribed `after*` event callbacks complete, you can invoke logic in registered Promise method callbacks that can modify any processing done by the event callbacks.

Note that your programming requirements for any `afterCreate`, `afterUpdate`, `afterDelete` (CUD operation) event callback, and for any `afterSaveChanges` event or Promise method callback can be affected by your setting of the JSDO `autoApplyChanges` property.

Behavior of event and Promise callbacks when autoApplyChanges is true

If you set the `autoApplyChanges` property to `true` (the default setting) before you invoke `saveChanges(false)`, and a corresponding record-change (CUD) operation succeeds on the server, the JSDO automatically accepts and synchronizes the change in JSDO memory.

If any CUD operation fails, the operation is automatically undone and the JSDO rejects the change by reverting the applicable record in JSDO memory to its last-saved state. Specifically:

- If a Create operation fails, the record is removed from JSDO memory.
- If an Update operation fails, the record reverts to the state it was in immediately preceding the `assign( )` method invocation that led to the failure.
- If a Delete operation fails, the record is restored to JSDO memory in its last-saved state. This state does not reflect any unsaved `assign( )` method invocations that may have occurred before the `remove( )` call.

When the JSDO synchronizes JSDO memory for CUD operations, it uses any before-image data (if available) in each response from invoking the `saveChanges( )` method.

If you invoke a Submit operation (`saveChanges(true)`) on an OpenEdge ProDataSet resource with before-image support, JSDO memory is automatically updated, with changes accepted or rejected based on the success or failure of each record create, update, and delete included in the Submit operation request. If the Submit returns with a network or server error that prevented the operation from being invoked on the server, all changes in JSDO memory associated with the Submit are rejected.

Note: For a Submit operation on a before-image resource, if you want all such record changes rejected if even one record change returns an error, set `autoApplyChanges` to `false` and explicitly reject all record changes based on the returned error condition. For more information on behavior when `autoApplyChanges` is `false`, see Behavior of event and Promise callbacks when autoApplyChanges is false on page 178 in this `saveChanges( )` method description.

When `autoApplyChanges` is `true`, the JSDO automatically clears any error conditions set for the affected record changes only after the record changes have all been rejected and undone and after all registered `after*` event and Promise method callbacks have executed. If any callback executes with unsuccessful results, you can call the JSDO `getErrors( )` method on a table reference to return any errors from the resource operation or operations associated with the specified JSDO table. For more information, see Error handling on page 179 in this `saveChanges( )` method description.

Note: The errors from CUD or Submit operations that you return using `getErrors( )` remain available for return until the next invocation of `fill( )` or `saveChanges( )`, regardless of the `autoApplyChanges` setting. For more information, see the getErrors( ) method on page 110 description.

Note: You can manually inspect error conditions and information as part of `after*` event and *`promise`*`.fail( )` callback execution when `autoApplyChanges` is `true`. However, checking for such error information might not be as useful as when `autoApplyChanges` is `false`, because all JSDO memory changes associated with record-change errors are automatically undone.

Caution: Use care when taking any action within event or Promise method callbacks that might interfere with the automatic acceptance or rejection of pending record changes. If you want to manually manage the handling of pending record changes in response to resource operations in these callbacks, consider setting `autoApplyChanges` to `false` before invoking `saveChanges( )`.

Behavior of event and Promise callbacks when autoApplyChanges is false

If you set the `autoApplyChanges` property to `false` before you invoke `saveChanges( )`, you must use one of the following methods to manually accept or reject any record changes in order to synchronize JSDO memory with operation results from the server. Use the method that is appropriate for the resource operation and the JSDO event or Promise method callback where you manage these changes:

- `acceptChanges( )`
- `acceptRowChanges( )`
- `rejectChanges( )`
- `rejectRowChanges( )`

Depending on the success of the particular resource operation and the JSDO `after*` event or Promise method callback in which you respond to operation results, you can check returned request object properties to determine what `accept*( )` or `reject*( )` method to call. If any callback executes with unsuccessful results, you can call the JSDO `getErrors( )` method on a table reference to return any errors from the resource operation or operations associated with the specified JSDO table.

Note: The errors from CUD or Submit operations that you return using `getErrors( )` remain available for return until the next invocation of `fill( )` or `saveChanges( )`, regardless of the `autoApplyChanges` setting. For more information, see the getErrors( ) method on page 110 description.

If you are handling results for one or more record-change (CUD) operations in an `afterSaveChanges` event callback, or in a Promise method callback, in response to calling `saveChanges(false)` (without using Submit), you can use the `batch` property of the request object to evaluate the results of each CUD operation invoked by `saveChanges( )`.

If you are handling results of a Submit operation (`saveChanges(true)`) on an OpenEdge ProDataSet resource with before-imaging, in an `afterSaveChanges` event or Promise method callback you can use the returned `jsrecords` property of the request object to help evaluate the results of each attempted record-change.

If you want to evaluate the results of any single record-change operation invoked by `saveChanges(false)` (a CUD operation with or without before-image support), or `saveChanges(true)` (a Submit operation) with before-image support, you can inspect the request object returned in an appropriate `after*` event callback that you have registered for each CUD operation, where you can identify the associated record change using the `jsrecord` property.

Note that the `acceptChanges( )` and `rejectChanges( )` methods might not be as useful for synchronizing JSDO memory with the server as the corresponding `acceptRowChanges( )` and `rejectRowChanges( )` methods, which you can invoke selectively in response to the results of individual record-change operations. The `acceptChanges( )` and `rejectChanges( )` methods are most useful for accepting or rejecting all record changes in JSDO memory based on the results of a single server transaction (invoked using Submit with before-image support) that either succeeds and applies all server record changes or fails if even one record change fails and undoes all server record changes as part of undoing the transaction. For more information, see the reference description for each method.

For more information on handling errors from failed record-change operations, see Error handling on page 179 in this `saveChanges( )` method description.

Error handling

You can use two mechanisms for retrieving information about any failure (error) in the most recently executed `saveChanges( )` call and the record changes (with or without before-image support) that might be discarded as a result:

- Call the `getErrors( )` method on any or all JSDO table references involved in the operation. For more information, see the getErrors( ) method on page 110 description.
- Access properties on the *`request`* parameter object returned by an event or Promise method callback that is executed in response to the operation. Use this mechanism to return error information for specific cases, for example, to return related data for an error using the `jsrecord` or `jsrecords` property. For more information, see the request object on page 37 description.

An error has occurred when the *`success`* parameter of the callback or the `success` property of the returned request object is `false`. (Note that for certain Promise method callbacks, such as those registered by *`promise`*`.done( )` and especially *`promise`*`.fail( )` for error handling, there is no need to check the value of the *`success`* parameter or `success` property passed to the callback, because the Promise method callback itself executes based on this value.)

The request object properties available for retrieving error information depend on whether the Data Object resource supports before-image (BI) data and on what `after*` event or Promise method callback is currently executing for the resource operation, or operations, that you are handling. Note that regardless of resource support for BI data, or the currently executing callback, the `getErrors( )` method always returns all error information associated with the most recently invoked CUD or Submit operations invoked as part of the most recent `saveChanges( )` call.

Note: These error handling options are most useful when the `autoApplyChanges` property is `false`. When `autoApplyChanges` is `true`, the method automatically accepts or rejects record changes as previously described for this property setting, and clears all associated error conditions and information either after the final `after*` event fires and executes its final subscribed callback for a given operation or after the final callback registered by a Promise method executes, whichever one executes last. Note that even after the errors are cleared, the `getErrors( )` method continues to return the errors associated with the most recently executed `saveChanges( )` call until the next `fill( )` or `saveChanges( )` call.

Thus, for a resource that:

- ***Does not support* before-image data with individual CUD record-change operations by `saveChanges(false)`** — To retrieve discarded changes and error information returned as part of the response to an:
  - **`after*` event fired for each CUD operation request invoked on a single record of data only** — Either call `getErrors( )` **or** inspect the `response` and `xhr` properties of the callback *`request`* parameter object for any error information. Inspect the callback *`record`* parameter or the `jsrecord` property (the record object) of the returned request object for information on operation data.
  - **`afterSaveChanges` event fired or a *`promise`*`.fail( )` method callback executed after all individual CUD operations on one or more records of data complete** — Call `getErrors( )` and inspect the `batch` property of the callback *`request`* parameter object to identify and retrieve any additional information and data returned for each CUD operation executed as part of invoking `saveChanges(false)`.
- ***Supports* before-image data with individual CUD record-change operations executed by `saveChanges(false)` (without using Submit)** — To retrieve discarded changes and error information returned as part of a before-image response to an:
  - **`after*` event fired for a specific CUD operation on a single record of data** — **Either** call `getErrors( )` and inspect the callback *`record`* parameter or the `jsrecord` property (the record object) of the returned request object for information on the operation data, or inspect the `response` and `xhr` properties of the callback *`request`* parameter object for any error information. If you inspect the `response` and `xhr` properties and no error information is found, inspect the callback *`record`* parameter or the `jsrecord` property (the record object) of the returned request object. As part of this inspection, check the return value of the `getErrorString( )` method called on the record object; a value other than `null` or `undefined` returns an error response for the corresponding before-image CUD operation executed on the server.
  - **`afterSaveChanges` event fired or a *`promise`*`.fail( )` method callback executed for CUD operations on one or more records of data** — Call `getErrors( )` and inspect the `batch` property of the callback *`request`* parameter object to identify and retrieve any additional information and data returned for each CUD operation executed as part of invoking `saveChanges(false)`.
- ***Supports* before-image data with a single Submit operation on one or more records of data executed by `saveChanges(true)`** — To retrieve discarded changes and error information returned as part of a before-image response from an:
  - **`after*` event fired for each record-change (CUD) as part of the single Submit operation** — Either call `getErrors( )` and inspect the callback *`record`* parameter or the `jsrecord` property (the record object) of the returned request object for information on the operation data, **or** inspect the `response` and `xhr` properties of the callback *`request`* parameter object for any error information on the specific record change. If you inspect the `response` and `xhr` properties and no error information is found, inspect the callback *`record`* parameter or `jsrecord` property (the record object) of the request object returned. As part of this inspection, check the return value of the `getErrorString( )` method called on the record object; a value other than `null` or `undefined` returns an error response for this record change as part of the Submit operation executed on the server.

- **`afterSaveChanges` event fired or a *`promise`*`.fail( )` method (or equivalent) callback executed for the single Submit operation** — Either call `getErrors( )` or inspect the `response` and `xhr` properties of the callback *`request`* parameter object for any error information on the entire Submit operation.

  If you call `getErrors( )` to return Submit operation errors, using any associated record ID information, you can also search the record objects returned by the `jsrecords` property of the callback *`request`* parameter object to identify corresponding record changes and inspect their before-image data.

  If you inspect the `response` and `xhr` properties, and no error information is found, inspect the `jsrecords` property to access all the record objects that were targets of the Submit operation and their corresponding record changes and before-image data. As part of this inspection, check the return value of the `getErrorString( )` method called on each record object; a value other than `null` or `undefined` returns an error response for this record change as part of the Submit executed on the server.

Examples

The following examples show the calling of `saveChanges( )` for various types of Data Object resources, with and without before-image support, and using different callback mechanisms to handle the results.

The following code fragment shows a JSDO created for an OpenEdge ProDataSet resource with before-imaging and its `autoApplyChanges` property set to `false` so it does not automatically accept or reject changes to data in JSDO memory after a call to the `saveChanges( )` method. Instead, it subscribes a single callback for each of the `afterDelete`, `afterCreate`, and `afterUpdate`, events to determine if changes to any `eCustomer` table record in JSDO memory should be accepted or rejected based on the success of the corresponding resource operation on the server. To change the data for a record, a jQuery event is defined on an update button to update the corresponding `eCustomer` record in JSDO memory with the current field values entered in a customer detail form (`#custdetail`):

Table 13: Example: saveChanges( ) using Submit with before-imaging that accepts/rejects changes based on CUD events

```
dataSet = new progress.data.JSDO( { name: 'dsCustomerOrder',
                                                autoApplyChanges : false } );
dataSet.eCustomer.subscribe('afterDelete', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterCreate', onAfterCustomerChange, this);
dataSet.eCustomer.subscribe('afterUpdate', onAfterCustomerChange, this);

$('#btnUpdate').bind('click', function(event) {
  var jsrecord = dataSet.eCustomer.findById($('#custdetail #id').val());
  if (jsrecord) {jsrecord.assign();};
});

// Similar controls might be defined to delete and create eCustomer records...

$('#btnCommit').bind('click', function(event) {
  dataSet.saveChanges(true); // Invokes the resource Submit operation
});

function onAfterCustomerChange(jsdo, record, success, request) {
  if (success)  {
    record.acceptRowChanges();
    // Perform other actions associated with accepting this record change
  }
  else
  {
    record.rejectRowChanges();
    // Perform other actions associated with rejecting this record change
  }
}
```

When the button is clicked, the event handler uses the `findById( )` method to find the original record with the matching internal record ID (`#id`) and invokes the `assign( )` method on `jsrecord` with an empty parameter list to update its fields in `eCustomer` with any new values entered into the form. You might define similar events and controls to delete the `eCustomer` record or add a new `eCustomer` record.

A jQuery event also defines a commit button that when clicked invokes the `saveChanges( )` method, passing `true` as the *`use-submit`* parameter value, to apply all pending changes in JSDO memory on the server in a single network request. Using this parameter, all pending record deletes, creates, and updates, including before-image data, are sent to the server in a single ProDataSet as input to a Submit operation. This operation processes all the changes for each record delete, create, or update on the server, storing the results in the same ProDataSet, including any errors, for output to the client in a single network response.

After the method completes, and all results have been returned to the client from the server, the appropriate event for each resource Delete, Create, or Update operation fires even though multiple changes can be sent using a single Submit operation. If the operation was successful, the event handler calls `acceptRowChanges( )` to accept the `eCustomer` record change associated with the event in JSDO memory. If the operation was not successful, the event handler calls `rejectRowChanges( )` to reject the `eCustomer` record change. An advantage of using an event to manually accept or reject a record change is that you can perform other actions associated with this particular change, such as creating a local log that describes the change or reports the error.

The following examples show similar processing of results on an OpenEdge resource, first using an `afterSaveChanges` event callback with and without before-imaging, then using equivalent Promise method callbacks for the resource without before-imaging. The final example shows Promise callback handling. The instantiation of the JSDO login session and the JSDO referenced by `myjsdo` is assumed.

The following code fragment subscribes the callback function, `onAfterSaveChanges`, to handle the `afterSaveChanges` event fired on the JSDO, `myjsdo` after invoking the Submit operation for an OpenEdge ProDataSet resource with a `ttCustomer` table resource and before-image support:

Table 14: Example: saveChanges( ) using Submit on an OpenEdge ProDataSet resource and handled using the afterSaveChanges event

```
/* subscribe to event */
myjsdo.subscribe( 'afterSaveChanges', onAfterSaveChanges );

/* some code that initiates multiple CUD operations before
   sending them to the server */
var newrec = myjsdo.add();

. . .

var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.remove();};

/* call to saveChanges() using Submit with event handling
   and with autoApplyChanges set to true by default */
myjsdo.saveChanges(true);

function onAfterSaveChanges( jsdo , success , request ) {
    var lenErrors,
        errors,
        errorType;

    if (success) {
        /* all record changes invoked by saveChanges()Submit succeeded */
        /* for example, accept all changes and redisplay records in the JSDO table */
        jsdo.foreach( function(jsrecord) {
            /* reference the record/field as jsrecord.data.<fieldName> . . . */

        });
    }
    else { // success = false
```

```
        /* handle Submit operation errors where either the server request
           or one or more record changes might have failed */
        console.log("Operation: Submit");
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) { /* Each error */
            console.log(JSON.stringify(errors[idxError]));
        } /* for each error message */
    }
};
```

In the above example, the `autoApplyChanges` property is set to `true` by default to allow `saveChanges( )` to save or reject all changes automatically based on the success of the Submit operation. If the method completes successfully, the JSDO simply accepts all changes in JSDO memory and re-displays the latest data from the records in the JSDO. If it completes unsuccessfully, the JSDO accepts or rejects each record change in JSDO memory based on the success of each submitted record change after the subscribed event callback has executed.

Note: For a Submit operation, you can determine the success of each record change in the `onAfterSaveChanges` callback based on the result of calling the JSDO `getErrorString( )` method on each record object that you return from the value of the `request.jsrecords` array property.

Also, if the method completes unsuccessfully, the event callback accesses any server exception information by calling the `getErrors( )` method on the single table in the JSDO (`jsdo` parameter). It concatenates any errors found and logs them to the console, possibly in preparation for having the user re-enter their changes. In this case, the error object returned by `getErrors( )` for each error is converted to a string in the form of a JSON object using the standard `JSON.stringify( )` method.

Note: These messages are accessible by `getErrors( )` until the next call to `fill( )` or `saveChanges( )`.

The following code fragment subscribes the callback function, `onAfterSaveChanges`, to handle the `afterSaveChanges` event fired on the JSDO, `myjsdo`, for an OpenEdge single-table resource, `ttCustomer`, without before-imaging:

Table 15: Example: saveChanges( ) on an OpenEdge resource without before-imaging using the afterSaveChanges event

```
/* subscribe to event */
myjsdo.subscribe( 'afterSaveChanges', onAfterSaveChanges );

/* some code that initiates multiple CUD operations before
   sending them to the server */
var newrec = myjsdo.add();

. . .

var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.remove();};

/* call to saveChanges() and event handling */
myjsdo.autoApplyChanges = false;
myjsdo.saveChanges();

function onAfterSaveChanges( jsdo , success , request ) {

    var operations = request.batch.operations,
        len = operations.length, /* number of resource operations invoked */
```

```
        lenErrors,
        errors,
        errorType;

    if (success) {
        /* all resource operations invoked by saveChanges() succeeded */
        /* for example, redisplay records in the JSDO table */
        jsdo.acceptChanges();
        jsdo.foreach( function(jsrecord) {
            /* reference the record/field as jsrecord.data.<fieldName> . . . */

        });
    }
    else {
        /* one or more resource operations invoked by saveChanges() failed */
        /* handle all operation error messages */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "Server Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
                case default:
                    errorType = null; // Unexpected errorType value
                    break;
            }
            if (errorType) { /* log all error text
                console.log("ERROR: " + errorType + errors[idxError].error);
                if (errors[idxError].id) { /* error with record object */
                    console.log("RECORD ID: " + errors[idxError].id);
                    /* possibly log the data values for record with this ID */
                }
                if (errors[idxError].responseText) {
                    console.log("HTTP FULL TEXT: "
                                + errors[idxError].responseText);
                }
            }
            else { /* unexpected errorType */
                console.log("UNEXPECTED ERROR TYPE: "
                            + errors[idxError].type);
            }
        } /* for each error message */

        /* accept or reject record changes based on each operation success */
        for(var idx = 0; idx < len; idx++) {
            var operationEntry = operations[idx];
            if (operationEntry.success) {
                operationEntry.jsrecord.acceptRowChanges();
            }
            else {
                operationEntry.jsrecord.rejectRowChanges();
            }
        } /*for each CUD operation */

    }
};
```

In the above example, the autoApplyChanges property is dynamically set to false prior to calling saveChanges( ) to allow custom handling of the results. If the method completes successfully, the callback simply accepts all changes in JSDO memory and re-displays the latest data from the records in the JSDO.

If the method completes unsuccessfully, the callback logs the error messages (jsdo.ttCustomer.getErrors()) returned for all the operations invoked by saveChanges( ) that failed. In this case, the property values from each error object returned by getErrors( ) are formatted and logged based on the error type for a more readable and informative output than a simple string representation of the JSON object. Finally, it accesses the batch property of the request object returned for the afterSaveChanges event in order to inspect the request object for each resource operation (operations[idx]) that was invoked by saveChanges( ). It then accepts or rejects the record changes for each operation record (operationEntry.jsrecord), depending on the operation success.

Note: These messages are accessible by getErrors( ) until the next call to fill( ) or saveChanges( ).

Note: For a Delete operation error, no data is available in operationEntry.jsrecord because the operation record has already been cleared from JSDO local memory. However, after executing operationEntry.jsrecord.rejectRowChanges( ) to undo the operation on the JSDO, the record is restored to JSDO local memory as expected.

The following code fragment registers done( ) and fail( ) method callbacks on the Promise returned by a saveChanges( ) call on a JSDO, myjsdo, which provides identical handling of operation results for the same OpenEdge resource handled in the previous example using the afterSaveChanges event callback:

Table 16: Example: saveChanges( ) on an OpenEdge resource using a Promise object

```
/* some code that initiates multiple CUD operations before
   sending them to the server */
var newrec = myjsdo.add();

. . .

var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.remove();};

/* call to saveChanges() with inline-coded Promise callback handling */
myjsdo.autoApplyChanges = false;
myjsdo.saveChanges().done(
    function( jsdo, success, request ) {
        /* all resource operations invoked by saveChanges() succeeded */
        /* for example, redisplay records in the JSDO table */
        jsdo.acceptChanges();
        jsdo.foreach( function(jsrecord) {
            /* reference the record/field as jsrecord.data.<field-ref> . . . */

        });
    }).fail(
    function( jsdo, success, request ) {
        /* one or more resource operations invoked by saveChanges() failed */
        var operations = request.batch.operations,
            len = operations.length, /* number of resource operations invoked */
            lenErrors,
            errors,
            errorType;

        /* handle all operation error messages */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
```

```
                    errorType = "Server Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
                case default:
                    errorType = null; // Unexpected errorType value
                    break;
            }
            if (errorType) { /* log all error text
                console.log("ERROR: " + errorType + errors[idxError].error);
                if (errors[idxError].id) { /* error with record object */
                    console.log("RECORD ID: " + errors[idxError].id);
                    /* possibly log the data values for record with this ID */
                }
                if (errors[idxError].responseText) {
                    console.log("HTTP FULL TEXT: "
                                + errors[idxError].responseText);
                }
            }
            else { /* unexpected errorType */
                console.log("UNEXPECTED ERROR TYPE: "
                            + errors[idxError].type);
            }
        } /* for each error message */

        /* accept or reject record changes based on each operation success */
        for(var idx = 0; idx < len; idx++) {
            var operationEntry = operations[idx];
            if (operationEntry.success) {
                operationEntry.jsrecord.acceptRowChanges();
            }
            else {
                operationEntry.jsrecord.rejectRowChanges();
            }
        } /*for each CUD operation */

    });
```

In the above example, the same processing occurs for successful and unsuccessful execution of `saveChanges( )`, but there is no need to test the `success` parameter of the `done( )` and `fail( )` Promise method callbacks, because they each execute in response to this value.

Note: For a Delete operation error, no data is available in `operationEntry.jsrecord` because the operation record has already been cleared from JSDO local memory. However, after executing `operationEntry.jsrecord.rejectRowChanges( )` to undo the operation on the JSDO, the record is restored to JSDO local memory as expected.

However, if your code already uses the `afterSaveChanges` event implementation, and you want to quickly change it to use Promises, you can simply register the original `onAfterSaveChanges` function as the callback for the `always( )` Promise method, as in the following code fragment:

Table 17: Example: saveChanges( ) using a Promise object that registers an afterSaveChanges callback

```
/* some code that initiates multiple CUD operations before
   sending them to the server */
var newrec = myjsdo.add();

. . .

var jsrecord = myjsdo.findById(myid);
if (jsrecord) {jsrecord.remove();};

/* call to saveChanges() with Promise callback handling using
   the original afterSaveChanges event handler */
myjsdo.autoApplyChanges = false;
myjsdo.saveChanges().always(onAfterSaveChanges);

function onAfterSaveChanges( jsdo , success , request ) {

    var operations = request.batch.operations,
        len = operations.length, /* number of resource operations invoked */
        lenErrors,
        errors,
        errorType;

    if (success) {
        /* all resource operations invoked by saveChanges() succeeded */
        /* for example, redisplay records in the JSDO table */
        jsdo.acceptChanges();
        jsdo.foreach( function(jsrecord) {
            /* reference the record/field as jsrecord.data.<fieldName> . . . */

        });
    }
    else {
        /* one or more resource operations invoked by saveChanges() failed */
        /* handle all operation error messages */
        errors = jsdo.ttCustomer.getErrors();
        lenErrors = errors.length;
        for (var idxError=0; idxError < lenErrors; idxError++) {
            switch(errors[idxError].type) {
                case progress.data.JSDO.DATA_ERROR:
                    errorType = "Server Data Error: ";
                    break;
                case progress.data.JSDO.RETVAL:
                    errorType = "Server App Return Value: ";
                    break;
                case progress.data.JSDO.APP_ERROR:
                    errorType = "Server App Error #"
                                + errors[idxError].errorNum + ": ";
                    break;
                case progress.data.JSDO.ERROR:
                    errorType = "Server General Error: ";
                    break;
                case default:
                    errorType = null; // Unexpected errorType value
                    break;
            }
            if (errorType) { /* log all error text
                console.log("ERROR: " + errorType + errors[idxError].error);
                if (errors[idxError].id) { /* error with record object */
                    console.log("RECORD ID: " + errors[idxError].id);
                    /* possibly log the data values for record with this ID */
                }
                if (errors[idxError].responseText) {
                    console.log("HTTP FULL TEXT: "
                                + errors[idxError].responseText);
                }
```

```
            }
            else { /* unexpected errorType */
                console.log("UNEXPECTED ERROR TYPE: "
                            + errors[idxError].type);
            }
        } /* for each error message */

        /* accept or reject record changes based on each operation success */
        for(var idx = 0; idx < len; idx++) {
            var operationEntry = operations[idx];
            if (operationEntry.success) {
                operationEntry.jsrecord.acceptRowChanges();
            }
            else {
                operationEntry.jsrecord.rejectRowChanges();
            }
        } /*for each CUD operation */

    }
};
```

Note: For a Delete operation error, no data is available in `operationEntry.jsrecord` because the operation record has already been cleared from JSDO local memory. However, after executing `operationEntry.jsrecord.rejectRowChanges( )` to undo the operation on the JSDO, the record is restored to JSDO local memory as expected.

Note: For a Delete operation error, no data is available in `operationEntry.jsrecord` because the operation record has already been cleared from JSDO local memory. However, after executing `operationEntry.jsrecord.rejectRowChanges( )` to undo the operation on the JSDO, the record is restored to JSDO local memory as expected.

See also:

acceptChanges( ) method on page 44, acceptRowChanges( ) method on page 47, autoApplyChanges property on page 82, batch property on page 85, data property on page 96, fill( ) method on page 97, getErrors( ) method on page 110, getErrorString( ) method on page 115, invocation method on page 134, jsrecord property on page 143, jsrecords property on page 144, rejectChanges( ) method on page 165, rejectRowChanges( ) method on page 168, response property on page 172, success property on page 202

saveLocal( ) method

Saves JSDO memory to a specified local storage area, including pending changes and any before-image data, according to a specified data mode.

Return type: `undefined`

Applies to: progress.data.JSDO class on page 12

Syntax

```
saveLocal ( [ storage-name [ , data-mode ] ] )
saveLocal ( data-mode )
```

storage-name

The name of the local storage area in which to save the specified data from JSDO memory. If *storage-name* is not specified, blank, or `null`, the name of the default storage area is used. The name of this default area is `jsdo_`*`serviceName`*`_`*`resourceName`*, where *serviceName* is the name of the Data Object Service that supports the JSDO instance, and *resourceName* is the name of the resource (table, dataset, etc.) for which the JSDO instance is created.

data-mode

A JSDO class constant that specifies the data in JSDO memory to be saved to local storage. Each data mode initially clears the specified local storage area of all its data, then replaces it with the data from JSDO memory as specified. The possible values include:

- **`progress.data.JSDO.ALL_DATA`** — Replaces the data in the storage area with **all** the data from JSDO memory, including pending changes and any before-image data. This is the default data mode.
- **`progress.data.JSDO.CHANGES_ONLY`** — Replaces the data in the storage area with **only** the pending changes from JSDO memory, including any before-image data.

If this method encounters any errors, it leaves the specified storage area unchanged and throws an exception.

This method supports any schema that the JSDO supports.

Note: If you want to save JSDO memory to local storage after the JSDO `saveChanges( )` method fails in response to an offline condition, be sure to set the `autoApplyChanges` property on the JSDO to `false` before calling this method for the first time. You can then continue to save JSDO memory to protect against a local session failure as it accumulates further offline changes until the mobile app goes back online and `saveChanges( )` succeeds in saving the changes to the server.

You can also use this method to routinely cache static data, such as state and rate tables, that might not change often, allowing for faster start up of the mobile app. One way to do this is to define a JSDO for a resource that accesses only static data, and invoke this method after refreshing JSDO memory using the `fill( )` method. When caching data in general, be sure to save JSDO memory when it is in a consistent state, such as immediately after successful invocation of the JSDO `fill( )` or `saveChanges( )` method (in the relatively rare case where routinely static data is being updated).

Example

The following code fragment caches JSDO memory for a JSDO defined with static data immediately after it is loaded into JSDO memory:

```
dataSet = new progress.data.JSDO( 'dsStaticData' );
dataSet.fill();
dataSet.saveLocal();
```

In this case, all the data in JSDO memory is cached to the default storage area.

See also:

acceptChanges( ) method on page 44, autoApplyChanges property on page 82, fill( ) method on page 97, readLocal( ) method on page 164, rejectChanges( ) method on page 165, saveChanges( ) method on page 174

services property

Returns an array of objects that identifies the Data Object Services that have been loaded for the current `JSDOSession` object and its web application.

Data type: Array of `Object`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 25

You load Data Object Services for a `JSDOSession` object by loading the corresponding Data Service Catalogs using the object's `addCatalog( )` method. You can load these Catalogs either before or after you log into a web application using the object's `login( )` method.

Each object in the array returned by this session property contains two properties:

- **`name`** — The name of a Data Object Service that is loaded for the specified session.
- **`uri`** — The URI for the service. If the address of the service in the Catalog is an absolute URI, this value is that URI. If the service address is relative, this value is the relative address concatenated to the value of the `JSDOSession` object's `serviceURI` property, which contains the web application URI used vy the `JSDOSession`.

Note: To return a corresponding list of URIs for the loaded Data Service Catalogs, read the `catalogURIs` property.

Example

Given two Catalogs, `CustomerSvc.json` and `ItemSvc.json`, that specify their services this way:

- **`"CustomerSvc"`** service with this URI: **`"/rest/CustomerSvc"`**
- **`"ItemSvc"`** service with this URI: **`"http://itemhost:8080/SportsApp/rest/ItemSvc"`**

The following code fragment produces the output that follows:

```
// create Session
                  pdsession = new
progress.data.JSDOSession('http://custhost:8080/SportsApp');

                  window.loginView = kendo.observable({
                  submit: function() {

                  // load 2 unprotected Catalogs
                  pdsession.addCatalog([

"http://custhost:8080/SportsApp/static/CustomerSvc.json","http://itemhost:8080/SportsApp/static/ItemSvc.json"
 ]).done(
                  function(JSDOsession, result, info) {

                  // log in anonymously
                  JSDOSession.login(
                    ).done(
                  function(JSDOSession, result, info) {

                  /* Use services property to print services
                  loaded by this Session object */
                  for (var i=0; i < JSDOSession.services.length; i++) {
                  console.log( JSDOSession.services[i].name + "   "
                  + JSDOSession.services[i].uri );
                  }
                  // Stuff with widgets and DataSources . . .
                  }).fail( // log in
                  function(JSDOSession, result, info) { . . .
                  });
                  }).fail( // Add Catalogs
                  function(JSDOSession, result, details) { . . .
                  });
                  }
                  });
```

Output from the preceding code fragment:

```
                  CustomerSvc   http://custhost:8080/SportsApp/rest/CustomerSvc
                  ItemSvc   http://itemhost:8080/SportsApp/rest/ItemSvc
```

See also:

addCatalog( ) method (JSDOSession class) on page 50, catalogURIs property on page 94, login( ) method (JSDOSession class) on page 145, serviceURI property on page 191

serviceURI property

Returns the URI to the web application that has been passed as an option to the class constructor for the current `JSDOSession` object.

Data type: `string`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 25

See also:

getSession( ) stand-alone function on page 121, login( ) method (JSDOSession class) on page 145

setProperties( ) method

Replaces all user-defined properties in the current JSDO instance with the user-defined properties defined in the specified object.

Return type: `null`

Applies to: progress.data.JSDO class on page 12

Syntax

```
setProperties ( propsObject )
```

propsObject

An `Object` containing a comma-separated list of *`name, value`* pairs that define the complete set of user-defined properties in the current JSDO, where *`name`* and *`value`* define a single property as defined for the `setProperty( )` method. The properties defined in *`propsObject`* entirely replace all other user-defined properties previously defined in the JSDO. You can also remove all existing user-defined properties from the JSDO by specifying `{}` as the value of *`propsObject`*.

Note: Any user-defined property you create with *`name`* set to `"server.count"` has a reserved usage in JSDO plugins. For more information, see the description of the `responseMapping` function that you can define using the addPlugin( ) method on page 57.

Example

The following code fragment sets the `"prop1"` user-defined property in `jsdoCustOrders` to the value `"12345"`. It then calls `setProperties( )` to replace all of the existing user-defined properties in `jsdoCustOrders` with definitions for the two user-defined properties, `"prop2"` and `"prop3"` and writes the current values of `"prop1"`, `"prop2"`, and `"prop3"` to the console log:

```
jsdoCustOrders = new progress.data.JSDO(. . .);

jsdoCustOrders.setProperty("prop1", "12345");
jsdoCustOrders.setProperties({"prop2", 100, "prop3", "ABCD"});
console.log("prop1: " + jsdoCustOrders.getProperty("prop1"));
console.log("prop2: " + jsdoCustOrders.getProperty("prop2"));
console.log("prop3: " + jsdoCustOrders.getProperty("prop3"));
```

The output from the `console.log` statements in this example appear as follows:

```
prop1: undefined
prop2: 100
prop3: ABCD
```

Note that the value returned and written out for "prop1" is undefined, because a user-defined property with that name is no longer defined in the JSDO.

See also:

getProperties( ) method on page 118, setProperty( ) method on page 193

setProperty( ) method

Sets the value of the specified JSDO user-defined property.

Return type: null

Applies to: progress.data.JSDO class on page 12

Syntax

```
setProperty ( name [ , value ] )
```

name

The name of a user-defined property to define in the current JSDO.

value

The value, if any, to set for the user-defined *name* property in the JSDO.

This method either defines a new user-defined property or updates an existing user-defined property definition in the JSDO. If *value* is not defined (or set to undefined), any existing *name* property definition is removed from the JSDO.

Note: Any user-defined property you create with *name* set to "server.count" has a reserved usage in JSDO plugins. For more information, see the description of the responseMapping function that you can define using the addPlugin( ) method on page 57.

Example

The following code fragment sets the "prop1" and "prop2" user-defined properties in jsdoCustOrders to the values, "12345" and 100, respectively:

```
jsdoCustOrders = new progress.data.JSDO(. . .);

jsdoCustOrders.setProperty("prop1", "12345");
jsdoCustOrders.setProperty("prop2", 100);
```

See also:

addPlugin( ) method on page 57, getProperty( ) method on page 119, setProperties( ) method on page 192

setSortFields( ) method

Specifies or clears the record fields on which to automatically sort the record objects for a table reference after you have set its `autoSort` property to `true` (the default).

This method enables or disables automatic sorting based on record fields only for supported JSDO operations. See the description of the `autoSort` property for more information.

Return type: `null`

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Syntax

```
jsdo-ref.setSortFields ( sort-fields )
jsdo-ref.table-ref.setSortFields ( sort-fields )
```

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference.

table-ref

A table reference on the JSDO.

sort-fields

An array of `string` values set to the names of record fields on which to sort the record objects, with an optional indication of the sort order for each field. This array can have the following syntax:

Syntax:

```
[ "field-name[:sort-order]" [ , "field-name[:sort-order]" ] ... ]
```

field-name

The name of a field in the record objects of the specified table reference. Any *field-name* must already exist in the JSDO schema and must have a scalar value (cannot be an array field).

sort-order

An indication of the sort order for the field, which can have one of the following case-insensitive values:

- `ASC` — Sorts ascending.
- `ASCENDING` — Sorts ascending.
- `DESC` — Sorts descending.
- `DESCENDING` — Sorts descending.

The default sort order is ascending.

When the automatic sort occurs, the record objects are sorted and grouped by each successive *field-name* in the array, according to its JavaScript data type and specified *sort-order*. Fields are compared using the >, <, and = JavaScript operators. All `string` fields can be compared with or without case sensitivity depending on the `caseSensitive` property setting. However, note that date fields are compared as dates, even though they are represented as strings in JavaScript.

If you set the *sort-fields* parameter to `null`, or you specify an empty array, the method clears all sort fields. Automatic sorting for the table reference can then occur only if there is an existing sort function setting using the `setSortFn( )` method.

Note: If you set a sort function for the table reference using `setSortFn( )` in addition to using this method to set sort fields, the sort function takes precedence.

Example

In the following code fragment, assuming the `autoSort` property is set to `true` on `dsCustomer.eCustomer` (the default), after the `fill( )` method initializes JSDO memory, the record objects for `eCustomer` are sorted by the `Country` field ascending, then by the `State` field within `Country` ascending, then by the `Balance` field within `State` descending. At a later point, the `foreach( )` method then loops through these record objects, starting with the first record in `eCustomer` sort order:

```
dsCustomer = new progress.data.JSDO( { name: 'dsCustomer' });
dsCustomer.eCustomer.setSortFields( [ "Country", "State", "Balance:DESC" ] );
dsCustomer.fill();
. . .
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );
```

See also:

autoSort property on page 82, caseSensitive property on page 93, setSortFn( ) method on page 195, sort( ) method on page 197

setSortFn( ) method

Specifies or clears a user-defined sort function with which to automatically sort the record objects for a table reference after you have set its `autoSort` property to `true` (the default).

This method enables or disables automatic sorting based on a sort function only for supported JSDO operations. See the description of the `autoSort` property for more information.

Return type: `null`

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Syntax

```
jsdo-ref.setSortFn ( funcRef )
jsdo-ref.table-ref.setSortFn ( funcRef )
```

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference.

table-ref

A table reference on the JSDO.

funcRef

A reference to a JavaScript sort function that compares two record objects for the sort and returns a `number` value. This function must have following signature:

Syntax:

```
function [ func-name ] ( jsrecord-ref1 , jsrecord-ref2 )
```

Where *func-name* is the name of a function that you define external to the `setSortFn( )` method parameter list, and *jsrecord-ref1* and *jsrecord-ref2* are two `JSRecord` objects that the function compares from the specified table reference. You can then pass *func-name* to the `setSortFn( )` method as the *funcRef* parameter. Alternatively, you can specify *funcRef* as the entire inline function definition without *func-name*.

Your *funcRef* code determines the criteria by which one of the two input record objects follows the other in the sort order, and returns one of the following values depending on the result:

- **1** — The *jsrecord-ref1* object follows (is "greater than") the *jsrecord-ref2* object in the sort order.
- **-1** — The *jsrecord-ref1* object precedes (is "less than") the *jsrecord-ref2* object in the sort order.
- **0** — The two record objects occupy the same position (are "equal") in the sort order.

When the JSDO invokes an automatic sort, and a sort function is set using this method, the sort uses this function to determine the sort order for every pair of records that it tests as it iterates through the record objects of the specified table reference.

If you set the *funcRef* parameter to `null`, the method clears any sort function definition. Automatic sorting for the table reference can then occur only if there are one or more existing sort fields set using the `setSortFields( )` method.

Note: Any default JavaScript comparisons that you make with `string` fields in *funcRef* are case sensitive according to JavaScript rules and ignore the setting of the `caseSensitive` property.

Note: If you set sort fields for the table reference using `setSortFields( )` in addition to using this method to set a sort function, the sort function takes precedence.

Examples

In the following code fragment, assuming the `autoSort` property is set to `true` on `dsCustomer.eCustomer` (the default), after the `fill( )` method initializes JSDO memory, the record objects for `eCustomer` are automatically sorted using the results of the external user-defined function, `sortOnNameCSensitive( )`, whose reference is passed to the `setSortFn( )` method. In this case, the function compares the case-sensitive values of the `Name` fields from each pair of `eCustomer` record objects selected by the sort. At a later point, the `foreach( )` method then loops through these record objects, starting with the first record in `eCustomer` sort order:

```
dsCustomer = new progress.data.JSDO( { name: 'dsCustomer' });
dsCustomer.setSortFn ( sortOnNameCSensitive );
dsCustomer.fill();
. . .
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );

function sortOnNameCSensitive ( rec1 , rec2 ) {
  if (rec1.data.Name > rec2.data.Name)
    return 1;
  else if (rec1.data.Name < rec2.data.Name)
    return -1;
  else
    return 0;
  }
```

If you want to compare the `Name` field in this function using a case-insensitive test, you can use the JavaScript `toUpperCase( )` function in the user-defined function. For example, in `sortOnNameCInsensitive( )`, as follows:

```
dsCustomer = new progress.data.JSDO( { name: 'dsCustomer' });
dsCustomer.setSortFn ( sortOnNameCInsensitive );
dsCustomer.fill();
. . .
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );

function sortOnNameCInsensitive ( rec1 , rec2 ) {
  if (rec1.data.Name.toUpperCase() > rec2.data.Name.toUpperCase())
    return 1;
  else if (rec1.data.Name.toUpperCase() < rec2.data.Name.toUpperCase())
    return -1;
  else
    return 0;
  }
```

See also:

autoSort property on page 82, caseSensitive property on page 93, setSortFields( ) method on page 194, sort( ) method on page 197

sort( ) method

Sorts the existing record objects for a table reference in JSDO memory using either specified sort fields or a specified user-defined sort function.

Return type: `null`

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Syntax

```
jsdo-ref.sort ( { sort-fields | funcRef } )
jsdo-ref.table-ref.sort ( { sort-fields | funcRef } )
```

jsdo-ref

A reference to the JSDO. You can call the method on *jsdo-ref* if the JSDO has only a single table reference.

table-ref

A table reference on the JSDO.

sort-fields

An array of `string` values set to the names of record fields on which to sort the record objects, with an optional indication of the sort order for each field. This array can have the following syntax:

Syntax:

```
[ "field-name[:sort-order]" [ , "field-name[:sort-order]" ] ... ]
```

field-name

The name of a field in the record objects of the specified table reference. Any *field-name* must already exist in the JSDO schema and must have a scalar value (cannot be an array field).

sort-order

An indication of the sort order for the field, which can have one of the following case-insensitive values:

- `ASC` — Sorts ascending.
- `ASCENDING` — Sorts ascending.
- `DESC` — Sorts descending.
- `DESCENDING` — Sorts descending.

The default sort order is ascending.

When the sort occurs, the record objects are sorted and grouped by each successive *field-name* in the array, according to its JavaScript data type and specified *sort-order*. Fields are compared using the >, <, and = JavaScript operators. All `string` fields can be compared with or without case sensitivity depending on the `caseSensitive` property setting. However, note that date fields are compared as dates, even though they are represented as strings in JavaScript.

funcRef

A reference to a JavaScript sort function that compares two record objects for the sort and returns a `number` value. This function must have following signature:

Syntax:

```
function [ func-name ] ( jsrecord-ref1 , jsrecord-ref2 )
```

Where *func-name* is the name of a function that you define external to the `sort( )` method parameter list, and *jsrecord-ref1* and *jsrecord-ref2* are two `JSRecord` objects that the function compares from the specified table reference. You can then pass *func-name* to the `sort( )` method as the *funcRef* parameter. Alternatively, you can specify *funcRef* as the entire inline function definition without *func-name*.

Your *funcRef* code determines the criteria by which one of the two input record objects follows the other in the sort order, and returns one of the following values depending on the result:

- **1** — The *jsrecord-ref1* object follows (is "greater than") the *jsrecord-ref2* object in the sort order.
- **-1** — The *jsrecord-ref1* object precedes (is "less than") the *jsrecord-ref2* object in the sort order.
- **0** — The two record objects occupy the same position (are "equal") in the sort order.

When you invoke the `sort( )` method with a sort function, the sort uses this function to determine the sort order for every pair of records that it tests as it iterates through the record objects of the specified table reference.

Note: Any default JavaScript comparisons that you make with `string` fields in *funcRef* are case sensitive according to JavaScript rules and ignore the setting of the `caseSensitive` property.

Caution: Because the `sort( )` method executes in JavaScript on the client side, sorting a large set of record objects can take a significant amount of time and make the UI appear to be locked. You might set a wait or progress indicator just prior to invoking the sort to alert the user that the app is working.

Examples

In the following code fragment, the `fill( )` method initializes JSDO memory with `eCustomer` record objects from the server in order of the table primary key (the default). The `sort( )` method later sorts the record objects for `eCustomer` by the `Country` field ascending, then by the `State` field within `Country` ascending, then by the `Balance` field within `State` descending. The `foreach( )` function then loops through these record objects in the new `eCustomer` sort order:

```
dsCustomer = new progress.data.JSDO( { name: 'dsCustomer' });
dsCustomer.fill();
. . .
dsCustomer.sort( [ "Country", "State", "Balance:DESC" ] );
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );
```

In the following code fragment, the `fill( )` method initializes JSDO memory with `eCustomer` record objects from the server in order of the table primary key (the default). The `sort( )` method later sorts the record objects for `eCustomer` using the results of an inline function definition, which in this case compares the case-sensitive values of the `Name` fields from each pair of `eCustomer` record objects selected by the sort. The `foreach( )` method then loops through these record objects in the new `eCustomer` sort order:

```
dsCustomer = new progress.data.JSDO( { name: 'dsCustomer' });
dsCustomer.fill();
. . .
dsCustomer.sort( function( rec1 , rec2 ) {
  if (rec1.data.Name > rec2.data.Name)
    return 1;
  else if (rec1.data.Name < rec2.data.Name)
    return -1;
  else
    return 0;
  } );
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );
```

If you want to compare the `Name` fields using a case-insensitive test, you can use the JavaScript `toUpperCase( )` function in the inline function definition, as follows:

```
dsCustomer = new progress.data.JSDO( { name: 'dsCustomer' });
dsCustomer.fill();
. . .
dsCustomer.sort( function( rec1 , rec2 ) {
  if (rec1.data.Name.toUpperCase() > rec2.data.Name.toUpperCase())
    return 1;
  else if (rec1.data.Name.toUpperCase() < rec2.data.Name.toUpperCase())
    return -1;
  else
    return 0;
  } );
dsCustomer.eCustomer.foreach( function( customer ){ . . . } );
```

See also:

autoSort property on page 82, caseSensitive property on page 93, setSortFields( ) method on page 194, setSortFn( ) method on page 195

subscribe( ) method (JSDO class)

Subscribes a given event callback function to a named event of the current JSDO or table reference.

For more information on these events, see the reference entry for progress.data.JSDO class on page 12.

Return type: `null`

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Syntax

```
jsdo-ref.subscribe ( event-name [ , op-name ] ,
                     event-handler [ , scope ] )
jsdo-ref.table-ref.subscribe ( event-name [ , op-name ] ,
                               event-handler [ , scope ] )
```

jsdo-ref

A reference to the JSDO. If you call the method on *jsdo-ref*, you can subscribe the event handler to any event that fires on the JSDO and all its table references.

table-ref

A table reference on the JSDO. If you call the method on *table-ref*, you can subscribe the event handler to an event that fires only on the table reference.

event-name

A `string` that specifies the name of an event to which you subscribe an event handler. See the reference entry for progress.data.JSDO class on page 12 for a list of available JSDO events.

If you call the `subscribe( )` method on *table-ref*, you can subscribe the event handler **only** to the following events:

- `afterCreate`
- `afterDelete`
- `afterUpdate`
- `beforeCreate`
- `beforeDelete`
- `beforeUpdate`

op-name

A `string` that specifies the name of a JSDO invocation method, a call to which causes the event to fire. This parameter is required in cases where *event-name* is `beforeInvoke` or `afterInvoke`. Use it **only** with these event names, and only when subscribing on the JSDO (not on a *table-ref*). The value of `op-name` is the same as the `fnName` property on the request object.

event-handler

A reference to an event handler function that is called when the specified event fires.

scope

An optional object reference that defines the execution scope of the event handler function called when the event fires. If the `scope` property is omitted, the execution scope is the global object (usually the browser or device window).

See also:

invocation method on page 134, progress.data.JSDO class on page 12, unsubscribe( ) method (JSDO class) on page 206

subscribe( ) method (JSDOSession class)

Subscribes a given event callback function to an event of the current `JSDOSession` object.

This method throws an exception if the specified event is not supported by the `JSDOSession` class.

Return type: `undefined`

Applies to: progress.data.JSDOSession class on page 25

Syntax

```
subscribe ( event-name , event-handler [ , scope ] )
```

`event-name`

A `string` that specifies the name of an event on a `JSDOSession` object to which you subscribe an event handler. See the reference entry for the progress.data.JSDOSession class on page 25 for a list of available events.

`event-handler`

A reference to an event handler function that is called when the specified event fires.

`scope`

An optional object reference that defines the execution scope of the event handler function called when the event fires. If the `scope` property is omitted, the execution scope is the global object (usually the browser or device window).

The `subscribe( )` method throws an error object if *`event-name`* does not identify an event supported by the `JSDOSession` class (the look up is not case sensitive), or if an argument is not of the correct type.

See also:

unsubscribe( ) method (JSDOSession class) on page 207

success property

A `boolean` that when set to `true` indicates that the Data Object resource operation was successfully executed.

This property is set from the HTTP status code returned from the server. A successful resource operation returns an HTTP status code in the range of 200 - 299, with one exception. An unsuccessful resource operation causes a value outside this range to be returned for the HTTP status code and sets this property to `false`. In addition, a record-change operation (Create, Update, Delete, or Submit) on before-image data that returns an HTTP status code of 200 OK, and returns a record object with a before-image error message is **also** unsuccessful, and sets this property to `false`.

Data type: `boolean`

Access: Read-only

Applies to: request object on page 37

The `success` property is available only for the following JSDO events or in the request object returned to a jQuery Promise callback:

- `afterCreate`
- `afterDelete`
- `afterFill`
- `afterInvoke`
- `afterSaveChanges`
- `afterUpdate`

In the case of an `afterSaveChanges` event, the `success` property is `true` only if **all** record-change operations that are invoked by the `saveChanges( )` method were successfully executed.

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe or register to JSDO events, any returned jQuery Promises, and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO. This object is also returned as the value of any JSDO invocation method that you execute synchronously.

Note: For an OpenEdge resource, when the server routine that implements a resource operation raises an unhandled error, this causes an HTTP status code of 500 and any included error information to be returned from the server. This can occur when either an ABL routine raises an application error or the ABL virtual machine (AVM) raises a system error, and the error is thrown out of the top-level server routine.

See also:

fill( ) method on page 97, invocation method on page 134, invoke( ) method on page 136, saveChanges( ) method on page 174

table reference property (JSDO class)

An object reference property on a JSDO that has the name of a corresponding table in the Data Object resource for which the current JSDO was created.

Its value is a reference (*table reference*) to the table object in JSDO memory. This table object provides access to a working record, if defined. If the server Data Object provides access to a multi-table resource, such as an OpenEdge ProDataSet, the JSDO provides one table reference for every table in the multi-table resource.

Data type: Table object reference in JSDO memory

Access: Read-only

Applies to: progress.data.JSDO class on page 12

In JSDO syntax descriptions, wherever a table reference can be used, for example, in *jsdo-ref*.*table-ref*, where *jsdo-ref* represents a reference to a JSDO instance, *table-ref* represents the name of a property containing the table reference. Every referenced table object, in turn, provides the following properties:

- **record property** — A reference to a JSRecord object (*record object*), which provides the data for the working record of the table in its own data property. This data property provides access to the field values of the working record as corresponding field reference properties (see the following bullet). If no working record is defined for the table object, its record property is null.
- **field reference property** — Also referred to as a *field reference*, a property on the table object that has the name and data type of a field (as defined in the table schema) and the value for that field in the working record. In JSDO syntax descriptions, wherever a field reference can appear, for example, in *jsdo-ref*.*table-ref*.*field-ref*, *field-ref* represents a property with the name of a corresponding table field containing the field reference.

  A JSDO table object provides one such field reference for each field defined in the corresponding resource table. If no working record is defined for a given table object, all of its field references are null. However, you can reference the current value for any *field-ref* on the data property of any JSRecord object reference that you have previously saved for a working record or that you otherwise return during a search of JSDO memory, for example, using the JSDO foreach( ) method.

  The JSDO also provides support for accessing array fields, which can be defined in an OpenEdge resource temp-table. You can reference individual array elements of any *field-ref* defined as an array using any of the following mechanisms:

  - **Standard JavaScript subscript reference** — Such as an integer subscript reference, *field-ref*[*index*], where *index* is a zero (0)-based integer expression that indexes the array.
  - **JSDO array-element reference** — Which specifies each element of the array as a separate field reference property with a unique name specified according to the format, *field-ref_integer*, where *field-ref* is the name of the original array field, *integer* is a one (1)-based integer that qualifies the field name, and where *field-ref* and *integer* are separated by a single underscore (_). Each *field-ref_integer* property contains the same value as the corresponding *field-ref* array element, *field-ref*[*index*], where *index* is equal to *integer* - 1. (**Note:** The *integer* name qualifier is one-based to match OpenEdge array field subscripts, which are one-based in OpenEdgeAdvanced Business Language (ABL).)

    Note: A given JSDO array-element reference can conflict with a scalar field that happens to have the same *field-ref_integer* name. If this happens for an OpenEdge resource table, you can resolve the conflict by assigning a different scalar field name to be generated in the Catalog for the corresponding ABL temp-table field definition. To assign a different generated field name, specify the SERIALIZE-NAME option for the affected scalar field in the temp-table definition. For more information, see the description of the DEFINE TEMP-TABLE statement in *OpenEdge Development: ABL Reference*.

You can therefore read field values in the working record of a given table reference using corresponding field references **either** on the data property of the JSRecord object returned by the record property of the table reference or directly on the table reference itself. However, although you can assign values directly to fields referenced on this data property, doing so does not trigger any automatic sorting of the data in JSDO memory, or any saving of the updated field value to the Data Object resource on the server. Instead, to write field values, Progress **strongly recommends** that you **either** call the JSDO assign( ) method on a *jsdo-ref*.*table-ref* with a working record (or on **any** valid JSRecord object reference), **or** assign values directly to any *jsdo-ref*.*table-ref*.*field-ref* with a working record.

Caution: Again, **never write directly** (by assigning a value) to a *field-ref* on the record.data property (or on the data property of any JSRecord object reference); use *field-ref* on this data property **only to read** the current field value. Writing field values directly on data.*field-ref* of a JSRecord object reference does **not** mark the record for update when you subsequently call the saveChanges( ) method, nor does it re-sort the updated record in JSDO memory according to any order you have established using the autoSort property. To mark a record for update and automatically re-sort the record according to the autoSort property, you **must** set a field value **either** by calling the assign( ) method on a record object **or** by assigning the value directly to the *jsdo-ref*.*table-ref*.*field-ref* of a working record, as described above.

Note: The record property with a working record does provide an alternative way to read a table field that happens to have the same name as any JSDO property (or method) that you can access (or invoke) directly on a table reference with a working record (**except**, of course, a table field with the name, record). That is, not every JSDO property (or method) that is available on a table reference is also available on the record.data property itself.

Examples

For example, the following code fragment shows two different ways to read the CustNum field of a record added to a Customer table provided by a Data Object resource ('CustomerOrderDS'):

```
var dataSet = new Progress.data.JSDO( 'CustomerOrderDS' );
dataSet.Customer.add();
alert(dataSet.Customer.record.data.CustNum);
alert(dataSet.Customer.CustNum);
```

Both calls to the alert( ) function access the same CustNum field in the working record of the Customer table created using the add( ) method.

In the following code fragment, MonthQuota is an array field with 12 elements (one for each month of the year) in the Salesrep table, and shows two different ways to read an element of the MonthQuota array from a record added to a Salesrep table provided by an OpenEdge Data Object resource (SalesDS):

```
var dataSet = new Progress.data.JSDO( 'SalesDS' );
var idx = 0;
dataSet.Salesrep.add();
alert(dataSet.Salesrep.record.data.MonthQuota[idx]);
alert(dataSet.Salesrep.MonthQuota_1);
```

Both calls to the alert( ) function access the same MonthQuota array element in the working record of the Salesrep table created using the add( ) method.

Note: Both of the previous examples assume the names of tables and fields in the OpenEdge sports2000 sample database. However, note also that where MonthQuota is presented as an array field for this example, that same field in the OpenEdge-installed instance of sports2000 is defined instead as a scalar integer field.

For more information on accessing the working record of a table reference, see the notes in the section on progress.data.JSDO class on page 12.

See also:

autoSort property on page 82, data property on page 96, progress.data.JSRecord class on page 35, record property on page 165

unsubscribe( ) method (JSDO class)

Unsubscribes a given event callback function from a named event of the current JSDO or table reference.

For more information on these events, see the refernce entry for progress.data.JSDO class on page 12.

Return type: `null`

Applies to: progress.data.JSDO class on page 12, table reference property (JSDO class) on page 203

Syntax

```
jsdo-ref.unsubscribe ( event-name [ , op-name ] ,
                       event-handler [ , scope ] )
jsdo-ref.table-ref.unsubscribe ( event-name [ , op-name ] ,
                                 event-handler [ , scope ] )
```

jsdo-ref

A reference to the JSDO. If you call the method on *jsdo-ref*, you can unsubscribe the event handler from any event that fires on the JSDO and all its table references.

table-ref

A table reference on the JSDO. If you call the method on *table-ref*, you can unsubscribe the event handler from an event that fires only on the table reference.

event-name

The name of an event to which you unsubscribe an event handler. See the refernce entry for progress.data.JSDO class on page 12 for a list of available JSDO events.

If you call the `unsubscribe( )` method on *table-ref*, you can unsubscribe the event handler **only** from the following events:

- `afterCreate`
- `afterDelete`
- `afterUpdate`
- `beforeCreate`
- `beforeDelete`
- `beforeUpdate`

op-name

The name of a JSDO invocation method, a call to which causes the event to fire. This parameter is required in cases where *event-name* is `beforeInvoke` or `afterInvoke`. Use it **only** with these event names, and only when unsubscribing on the JSDO (not on a *table-ref*). To work, the *op-name* value must match the corresponding *op-name* parameter in a preceding `subscribe( )` method call.

event-handler

A reference to an event handler function that is called when the specified event fires.

scope

An optional object reference that defines the execution scope of the event handler function called when the event fires. Specifying the scope is optional in the event subscription. If the event subscription **does** specify an execution scope, you must specify a matching *scope* parameter when you call the `unsubscribe( )` method to cancel the event subscription.

See also:

invocation method on page 134, progress.data.JSDO class on page 12, subscribe( ) method (JSDO class) on page 200, unsubscribeAll( ) method on page 208

unsubscribe( ) method (JSDOSession class)

Unsubscribes a given event callback function from an event of the current `JSDOSession` object.

This method throws an exception if the specified event is not supported by the `JSDOSession` class.

Return type: `undefined`

Applies to: progress.data.JSDOSession class on page 25

Syntax

```
unsubscribe ( event-name , event-handler [ , scope ] )
```

event-name

The name of a `JSDOSession` object event to which you unsubscribe an event handler. See the reference entry for the progress.data.JSDOSession class on page 25 for a list of available events.

event-handler

A reference to an event handler function that is to be removed from the list of callbacks that are called when the specified event fires.

scope

An optional object reference that defines the execution scope of the event handler function. Specifying the scope is optional in the event subscription. If the event subscription **does** specify an execution scope, you must specify a matching *scope* parameter when you call the `unsubscribe( )` method to cancel the event subscription.

The `unsubscribe( )` method throws an error object if *event-name* does not identify an event supported by the `JSDOSession` class (the look up is not case sensitive), if an argument is not of the correct type, or if there is no handler that was subscribed for that event with the same *scope* as passed to `unsubscribe( )`.

See also:

progress.data.JSDOSession class on page 25, subscribe( ) method (JSDOSession class) on page 202

unsubscribeAll( ) method

Unsubscribes all event callback functions from a single named event of the current JSDO, `JSDOSession` or `Session` object, or unsubscribes all event callback functions from all events of the current JSDO, `JSDOSession`, or `Session` object.

Return type: `null`

Applies to: progress.data.JSDO class on page 12, progress.data.JSDOSession class on page 25

Syntax

```
unsubscribeAll ( [ event-name ] )
```

`event-name`

A `string` that if specified, is the name of an event on the current object from which to unsubscribe all event handlers. If not specified, the method unsubscribes all event handlers from all events of the current object. See the reference entry for the progress.data.JSDO class on page 12 or the progress.data.JSDOSession class on page 25 for a list of available events.

For a `JSDOSession` or `Session` object, the `unsubscribeAll( )` method throws an error object if *`event-name`* does not identify an event supported by the `progress.data.JSDOSession` or `progress.data.Session` class (the lookup is case insensitive), or if *`event-name`* is not a `string`. For a JSDO instance, the method ignores these conditions.

See also:

unsubscribe( ) method (JSDO class) on page 206, unsubscribe( ) method (JSDOSession class) on page 207

useRelationships property

A `boolean` that specifies whether JSDO methods that operate on table references in JSDO memory recognize and honor *data-relations* defined in the schema (that is, work only on records of a child table that are related to the working record of a parent table).

Data type: `boolean`

Access: Readable/Writable

Applies to: progress.data.JSDO class on page 12

When set to `true`, methods, such as `add( )`, `find( )`, and `foreach( )`, that have default behavior for related table references respect these relationships when operating on related tables. When set to `false`, these methods operate on all table references as if they have no relationships. The default value is `true`.

Note: Data-relations defined for OpenEdge Data Object resources (ProDataSets) that use the `RECURSIVE` key word have no effect and are not supported by the JSDO.

See also:

add( ) method on page 49, find( ) method on page 104, foreach( ) method on page 108

userName property

Returns the username passed as a parameter to the most recent call to the `login( )` method on the current `JSDOSession` object.

Data type: `string`

Access: Read-only

Applies to: progress.data.JSDOSession class on page 25

This value is returned, whether or not the most recent call to `login( )` succeeded.

See also:

getSession( ) stand-alone function on page 121, login( ) method (JSDOSession class) on page 145

xhr property

A reference to the XMLHttpRequest object used to make an operation request on a resource of a Data Object Service.

In the case of an asynchronous call, this property may not be available until after the XMLHttpRequest object is created.

Note: You can use the XMLHttpRequest object referenced by this property to return error information that originates from the web server that hosts the Data Object Service handling the operation request. As of Progress Data Objects 4.3 and later, the JSDO provides a `getErrors( )` method that returns all error information for an operation request, including error information that originates from the hosting web server. For more information, see the getErrors( ) method on page 110 description.

Data type: `Object`

Access: Read-only

Applies to: request object on page 37

The `xhr` property is available for the following events or in the request object returned to a jQuery Promise callback:

- `afterFill`
- `afterInvoke`

The `xhr` property is available for the following events after calling the JSDO `saveChanges( )` method either with an empty parameter list or with the single parameter value of `false`:

- `afterCreate`

- `afterDelete`
- `afterUpdate`

The `xhr` property is available **only** for the following event or in the request object returned to a jQuery Promise callback after calling `saveChanges(true)` on a JSDO resource that supports a Data Object Submit operation:

- `afterSaveChanges`

This request object property is also available for any session `online` and `offline` events that are fired in response to the associated resource operation when it encounters a change in the online status of the JSDO login session (`JSDOSession` or `Session` object). The request object is itself passed as a parameter to any event handler functions that you subscribe or register to JSDO events, any returned jQuery Promises, and to the `online` and `offline` events of the session that manages Data Object Services for the JSDO. This object is also returned as the value of any JSDO invocation method that you execute synchronously.

See also:

fill( ) method on page 97, getErrors( ) method on page 110, invocation method on page 134, invoke( ) method on page 136, response property on page 172, saveChanges( ) method on page 174

Data type mappings for Data Object Services

Each type of Data Object Service maps the data types used and understood by its resources to JSON types for transport over the network, and the data are then converted to corresponding JavaScript data types when loaded into JSDO memory. A similar process happens in reverse when the JSDO sends data back over the network to the Data Object Service. The following topics provide an overview of JavaScript data type support in the JSDO and the supported mappings to the data types of OpenEdge ABL.

JavaScript data type overview

JavaScript supports five primitive data types, as shown in the following table.

Table 18: JavaScript primitive data types

Data type	Description	Examples
`string`	A string of characters enclosed in double or single quotes.	`"jump rope"`

`number`	An unquoted numeric value, which can include an exponent using scientific notation.	`17`
		`54.35`
		`0.9582e-42`

`boolean`	The unquoted, lowercase, literal value `true` or `false`.	`true`
`null`	The unquoted, lowercase, literal value, `null`.	`null`
`undefined`	The unquoted, lowercase, literal value, `undefined`.	`undefined`

The data type of a primitive value is determined by the format of the value:

- A string of characters surrounded by quotes indicates the value is a `string`.
- A string of numeric characters without surrounding quotes and with an optional decimal point, an optional negative sign, or an optional exponent indicates a `number`.
- The string `true` or `false` without surrounding quotes indicates a `boolean`.
- The string `null` without surrounding quotes indicates a literal `null` value that typically represents a variable, object property, or function return value that is not assigned to any other data type value.
- The string `undefined` without surrounding quotes also indicates a literal `undefined` value that typically represents a variable, object property, or function return value that is undefined.

In addition to these standard primitive data types, there are some commonly-used, but non-standard data types for certain values that are not officially supported in JavaScript. For these non-standard types, specially formatted strings represent values for which there is no standard primitive JavaScript data type.

The following table shows non-standard JavaScript data types that OpenEdge Data Object resources support.

Table 19: OpenEdge-supported non-standard JavaScript data types

Non-standard JavaScript data type	Representation
Date	A `string` in the ISO 8601 format, `"yyyy-mm-ddThh:mm:ss.sss+hh:mm"`. JavaScript does support a `Date` object for working with dates and times. However, all dates and times returned from an OpenEdge application server to a JSDO are stored as a `string` in the ISO 8601 format.
Binary data	A `string` consisting of the Base64 encoded equivalent of the binary data, which in OpenEdge can be represented in an ABL `BLOB`, `MEMPTR`, or `ROWID` data type.

For more information on the OpenEdge usage of the non-standard JavaScript data types in the previous table, see OpenEdge ABL to JavaScript data type mappings on page 212.

JavaScript also supports two complex data types, used to aggregate values of all JavaScript data types, including both primitive and complex data, as shown in the following table:

Table 20: JavaScript complex data types

Data type	Description	Examples

`Object`	A comma-delimited list of named values (*properties*), either primitive or complex, enclosed in braces ({}). The property names can either be literal values or quoted strings.	`{ myString : "jump rope",`
		`  'myNum' : 17,`
		`  'myBool' : false }`

`Array`	A comma-delimited list of unnamed values, either primitive or complex, enclosed in brackets ([])	`[ "jump rope", 17, false ]`

Note: JavaScript also supports standard objects with the same type names as the primitive data types written by convention using initial upper case, for example, a `Number` object. These objects serve as wrappers for the corresponding primitive types and provide additional operations on these primitive types.

OpenEdge ABL to JavaScript data type mappings

A JSDO communicates with an OpenEdge application server through a Data Object Service to marshal ABL data for parameters (and return values) of ABL routines that pass the following ABL data types: ProDataSets, temp-tables, arrays, and primitive values. The JSDO provides this ABL data to the mobile app programmer as JavaScript data. For more information on ABL data types, see the reference entry on data types in *OpenEdge Development: ABL Reference*. For more information on the format and use of JavaScript data types, you can review sources on the web, such as http://www.w3schools.com/js/js_datatypes.asp. For more information on JSDOs and how they communicate with an OpenEdge application server through a Data Object Service, see progress.data.JSDO class on page 12.

The following table shows the mappings between the supported ABL data types in an OpenEdge Data Object resource and the JavaScript data types of corresponding table fields and invocation method parameter values in a JSDO. These data type conversions happen in the server-side Data Object when the JSDO invokes a resource operation that exchanges data between the JSDO and its OpenEdge resource using the `fill( )`, `saveChanges( )`, or any invocation method.

Note: An ABL data item or Invoke routine output parameter that is set to the Unknown value (?) maps to the JavaScript `null` value when sent from the OpenEdge Data Object resource to the JSDO, and a JSDO data item or invocation method input parameter value that is set to `null` maps to the ABL Unknown value (?) when sent from the JSDO to the OpenEdge Data Object resource. The ABL `BLOB` and `CLOB` are only allowed as fields of temp-table parameters, and their respective equivalents, `MEMPTR` and `LONGCHAR`, are only allowed as scalar parameters or as elements of `Array` parameters.

Table 21: OpenEdge ABL data type mappings

ABL data type	JavaScript data type
BLOB[1]	string (Base64 encoded)
CHARACTER	string
CLOB[1]	string
COM-HANDLE	number
DATASET[2]	An Object that maps to a ProDataSet and contains one or more Object instances, each of which maps to an ABL temp-table in the ProDataSet (see TEMP-TABLE)
DATE[3]	string (ISO 8601 formatted string of the form "yyyy-mm-dd")
DATETIME[3]	string (ISO 8601 formatted string of the form "yyyy-mm-ddThh:mm:ss.sss")
DATETIME-TZ[3]	string (ISO 8601 formatted string of the form "yyyy-mm-ddThh:mm:ss.sss+hh:mm")
DECIMAL	number
primitive EXTENT[2]	Where *primitive* is an ABL primitive data type (not a DATASET or TEMP-TABLE), maps to an Array of the JavaScript primitive data type that maps to the corresponding ABL primitive data type
HANDLE	number
INT64	number
INTEGER	number
LOGICAL	boolean (true or false)
LONGCHAR[4]	string
MEMPTR[4]	string (Base64 encoded)
RAW	Not supported
RECID	Not supported

[1] In temp-tables only.

[2] ABL ProDataSets, temp-tables, and arrays map to JavaScript objects and arrays using a structure that is identical to the OpenEdge-supported mapping to JSON objects and arrays. JSON (JavaScript Object Notation) is a character representation of JavaScript data types that is often used as a lightweight alternative to XML. For more information on ABL support for JSON, including the JSON representation of these ABL data types, see *OpenEdge Development: Working with JSON*.

[3] See Date and time conversions on page 214.

[4] As scalar parameters or elements of Array parameters only.

ABL data type	JavaScript data type
ROWID	string (Base64 encoded)
TEMP-TABLE[2]	An Object that contains a single Array of Object instances, where each Object in the Array maps to a record in the corresponding temp-table

Date and time conversions

In the JSDO, all ABL DATE, DATETIME, and DATETIME-TZ data types are mapped to a string in the ISO 8601 format shown in the previous table. This format supports all the necessary functionality for these data types (date, time, and time zone data). However, if you read and convert (**Output**) this string from the JSDO to a JavaScript Date object, or write (**Input**) the value of a JavaScript Date object to the JSDO as a similarly ISO 8601-formatted string, the JSDO processes the value differently based on the original ABL data type in the OpenEdge resource, as shown in the following table:

Table 22: JavaScript Date object conversions to and from ISO 8601 strings

ABL data type	Process
DATE	**Output:** Client time zone is used. Time is set to 0 (midnight). **Input:** Time and time zone information is ignored.
DATETIME	**Output:** Client time zone is used. **Input:** Time zone information is ignored
DATETIME-TZ	**Output and Input:** Date, time, and time zone is preserved.

Note: This same processing occurs whether for reading and writing data on the OpenEdge server as part of Data Object Read (**Output**), Create (**Input**), and Update (**Input**) operations, or for reading and writing data on the server through the parameters (**Output** and **Input**) and return values (**Output**) of Invoke operations.

Note: This date and time conversion supports Kendo UI clients that access the JSDO as a remote data service using the JSDO dialect of the Kendo UI DataSource, which works with JSDO date and time data as JavaScript Date objects.

3

Data type mappings for Data Object Services

Each type of Data Object Service maps the data types used and understood by its resources to JSON types for transport over the network, and the data are then converted to corresponding JavaScript data types when loaded into JSDO memory. A similar process happens in reverse when the JSDO sends data back over the network to the Data Object Service. The following topics provide an overview of JavaScript data type support in the JSDO and the supported mappings to the data types of OpenEdge ABL.

For details, see the following topics:

- JavaScript data type overview
- OpenEdge ABL to JavaScript data type mappings

JavaScript data type overview

JavaScript supports five primitive data types, as shown in the following table.

Table 23: JavaScript primitive data types

Data type	Description	Examples
`string`	A string of characters enclosed in double or single quotes.	`"jump rope"`

`number`	An unquoted numeric value, which can include an exponent using scientific notation.	`17`
		`54.35`
		`0.9582e-42`

`boolean`	The unquoted, lowercase, literal value `true` or `false`.	`true`
`null`	The unquoted, lowercase, literal value, `null`.	`null`
`undefined`	The unquoted, lowercase, literal value, `undefined`.	`undefined`

The data type of a primitive value is determined by the format of the value:

- A string of characters surrounded by quotes indicates the value is a `string`.
- A string of numeric characters without surrounding quotes and with an optional decimal point, an optional negative sign, or an optional exponent indicates a `number`.
- The string `true` or `false` without surrounding quotes indicates a `boolean`.
- The string `null` without surrounding quotes indicates a literal `null` value that typically represents a variable, object property, or function return value that is not assigned to any other data type value.
- The string `undefined` without surrounding quotes also indicates a literal `undefined` value that typically represents a variable, object property, or function return value that is undefined.

In addition to these standard primitive data types, there are some commonly-used, but non-standard data types for certain values that are not officially supported in JavaScript. For these non-standard types, specially formatted strings represent values for which there is no standard primitive JavaScript data type.

The following table shows non-standard JavaScript data types that OpenEdge Data Object resources support.

Table 24: OpenEdge-supported non-standard JavaScript data types

Non-standard JavaScript data type	Representation
Date	A `string` in the ISO 8601 format, `"yyyy-mm-ddThh:mm:ss.sss+hh:mm"`. JavaScript does support a `Date` object for working with dates and times. However, all dates and times returned from an OpenEdge application server to a JSDO are stored as a `string` in the ISO 8601 format.
Binary data	A `string` consisting of the Base64 encoded equivalent of the binary data, which in OpenEdge can be represented in an ABL `BLOB`, `MEMPTR`, or `ROWID` data type.

For more information on the OpenEdge usage of the non-standard JavaScript data types in the previous table, see OpenEdge ABL to JavaScript data type mappings on page 212.

JavaScript also supports two complex data types, used to aggregate values of all JavaScript data types, including both primitive and complex data, as shown in the following table:

Table 25: JavaScript complex data types

Data type	Description	Examples

`Object`	A comma-delimited list of named values (*properties*), either primitive or complex, enclosed in braces ({}). The property names can either be literal values or quoted strings.	`{ myString : "jump rope",`
		`  'myNum' : 17,`
		`  'myBool' : false }`

`Array`	A comma-delimited list of unnamed values, either primitive or complex, enclosed in brackets ([])	`[ "jump rope", 17, false ]`

Note: JavaScript also supports standard objects with the same type names as the primitive data types written by convention using initial upper case, for example, a `Number` object. These objects serve as wrappers for the corresponding primitive types and provide additional operations on these primitive types.

OpenEdge ABL to JavaScript data type mappings

A JSDO communicates with an OpenEdge application server through a Data Object Service to marshal ABL data for parameters (and return values) of ABL routines that pass the following ABL data types: ProDataSets, temp-tables, arrays, and primitive values. The JSDO provides this ABL data to the mobile app programmer as JavaScript data. For more information on ABL data types, see the reference entry on data types in *OpenEdge Development: ABL Reference*. For more information on the format and use of JavaScript data types, you can review sources on the web, such as http://www.w3schools.com/js/js_datatypes.asp. For more information on JSDOs and how they communicate with an OpenEdge application server through a Data Object Service, see progress.data.JSDO class on page 12.

The following table shows the mappings between the supported ABL data types in an OpenEdge Data Object resource and the JavaScript data types of corresponding table fields and invocation method parameter values in a JSDO. These data type conversions happen in the server-side Data Object when the JSDO invokes a resource operation that exchanges data between the JSDO and its OpenEdge resource using the `fill( )`, `saveChanges( )`, or any invocation method.

Note: An ABL data item or Invoke routine output parameter that is set to the Unknown value (?) maps to the JavaScript `null` value when sent from the OpenEdge Data Object resource to the JSDO, and a JSDO data item or invocation method input parameter value that is set to `null` maps to the ABL Unknown value (?) when sent from the JSDO to the OpenEdge Data Object resource. The ABL `BLOB` and `CLOB` are only allowed as fields of temp-table parameters, and their respective equivalents, `MEMPTR` and `LONGCHAR`, are only allowed as scalar parameters or as elements of `Array` parameters.

Table 26: OpenEdge ABL data type mappings

ABL data type	JavaScript data type
BLOB[5]	string (Base64 encoded)
CHARACTER	string
CLOB[5]	string
COM-HANDLE	number
DATASET[6]	An Object that maps to a ProDataSet and contains one or more Object instances, each of which maps to an ABL temp-table in the ProDataSet (see TEMP-TABLE)
DATE[7]	string (ISO 8601 formatted string of the form "yyyy-mm-dd")
DATETIME[7]	string (ISO 8601 formatted string of the form "yyyy-mm-ddThh:mm:ss.sss")
DATETIME-TZ[7]	string (ISO 8601 formatted string of the form "yyyy-mm-ddThh:mm:ss.sss+hh:mm")
DECIMAL	number
primitive EXTENT[6]	Where *primitive* is an ABL primitive data type (not a DATASET or TEMP-TABLE), maps to an Array of the JavaScript primitive data type that maps to the corresponding ABL primitive data type
HANDLE	number
INT64	number
INTEGER	number
LOGICAL	boolean (true or false)
LONGCHAR[8]	string
MEMPTR[8]	string (Base64 encoded)
RAW	Not supported
RECID	Not supported

[5] In temp-tables only.

[6] ABL ProDataSets, temp-tables, and arrays map to JavaScript objects and arrays using a structure that is identical to the OpenEdge-supported mapping to JSON objects and arrays. JSON (JavaScript Object Notation) is a character representation of JavaScript data types that is often used as a lightweight alternative to XML. For more information on ABL support for JSON, including the JSON representation of these ABL data types, see *OpenEdge Development: Working with JSON*.

[7] See Date and time conversions on page 219.

[8] As scalar parameters or elements of Array parameters only.

ABL data type	JavaScript data type
ROWID	string (Base64 encoded)
TEMP-TABLE[6]	An Object that contains a single Array of Object instances, where each Object in the Array maps to a record in the corresponding temp-table

Date and time conversions

In the JSDO, all ABL DATE, DATETIME, and DATETIME-TZ data types are mapped to a string in the ISO 8601 format shown in the previous table. This format supports all the necessary functionality for these data types (date, time, and time zone data). However, if you read and convert (**Output**) this string from the JSDO to a JavaScript Date object, or write (**Input**) the value of a JavaScript Date object to the JSDO as a similarly ISO 8601-formatted string, the JSDO processes the value differently based on the original ABL data type in the OpenEdge resource, as shown in the following table:

Table 27: JavaScript Date object conversions to and from ISO 8601 strings

ABL data type	Process
DATE	**Output:** Client time zone is used. Time is set to 0 (midnight). **Input:** Time and time zone information is ignored.
DATETIME	**Output:** Client time zone is used. **Input:** Time zone information is ignored
DATETIME-TZ	**Output and Input:** Date, time, and time zone is preserved.

Note: This same processing occurs whether for reading and writing data on the OpenEdge server as part of Data Object Read (**Output**), Create (**Input**), and Update (**Input**) operations, or for reading and writing data on the server through the parameters (**Output** and **Input**) and return values (**Output**) of Invoke operations.

Note: This date and time conversion supports Kendo UI clients that access the JSDO as a remote data service using the JSDO dialect of the Kendo UI DataSource, which works with JSDO date and time data as JavaScript Date objects.